Дата последней сборки документа:
7 марта 2023 г. в 02:57

Конспект по книге Гудфеллоу «Глубокое обучение»*

Содержание

*Гудфеллоу Я., Бенджио И., Курвилль А. Глубокое обучение. – М.: ДМК Пресс, 2018. – 652 с.

1. Численные методы

2. Основы машинного обучения

2.1. Точечная оценка

Точечное оценивание – это попытка найти единственное «наилучшее» предсказание интересующей величины. Пусть $\{x^{(1)}, \ldots, x^{(m)}\}$ – множество m независимых и одинаково распределенных точек. *Точечной оценкой*, или *статистикой*, называется любая функция этих данных

$$\theta_m = g(x^{(1)}, \ldots, x^{(m)}).$$

В этом определении не требуется, чтобы g возвращала значение, близкое к истинному значению θ, ни даже чтобы область значений g совпадала со множеством допустимых значений θ.

Алгоритм k-групповой перекрестной проверки применяется для оценивания ошибки обобщения алгоритма обучения A, когда имеющийся набор данных $\mathbb{D}$ *слишком мал* для того, чтобы простое разделение на обучающий и тестовый или обучающий и контрольный наборы могло дать точную оценку ошибки обобщения, поскольку среднее значение потери L на малом тестовом наборе может иметь высокую дисперсию.

2.2. Смещение

Смещение оценки определяется следующим образом

$$\text{bias}(\hat{\theta}_m) = \mathbb{E}(\hat{\theta}_m) - \theta,$$

где математической ожидание вычисляется по данным (рассматривается как выборка из случайной величины), а θ – истинное значение параметра, которое определяет порождающее распределение.

Оценка $\hat{\theta}$ называется *несмещенной*, если

$$(\hat{\theta}_m) = 0, \text{ т.е. } \mathbb{E}(\hat{\theta}_m) = \theta.$$

Оценка $\hat{\theta}_m$ называется *асимптотически несмещенной*, если

$$\lim_{m\to\infty} \text{bias}(\hat{\theta}_m) = 0, \text{ т.е. } \lim_{m\to\infty} \mathbb{E}(\hat{\theta}_m) = \theta.$$

2.3. Дисперсия

Для определения смещения мы вычисляли математичесвкое ожидание оценки, но точно так же можем вычислить и ее дисперсию. *Дисперсией оценки* называется выражение

$$\text{Var}(\hat{\theta}).$$

Стандартной ошибкой $\text{SE}(\hat{\theta})$ называется квадратный корень из дисперсии.

Воспользовавшись центральной предельной теоремой, согласно которой среднее имеет приблизительно нормальное распределение, можем применить стандартную ошибку для вычисления вероятности того, что истинное математическое ожидание находится в выбранном интервале. Например, *95-процентный доверительный интервал* вокруг выборочного среднего (вокру оценки) $\hat{\mu}_m = \frac{1}{m}\sum_{k=1}^{n} x^{(i)}$ определяется формулой

$$(\hat{\mu}_m - 1.96\,\text{SE}(\hat{\mu}_m), \hat{\mu}_m + 1.96\,\text{SE}(\hat{\mu}_m))$$

при нормальном распределении со средним $\hat{\mu}_m$ и дисперсией $\text{SE}(\hat{\mu}_m)^2$.

NB: В экспериментах по машинному обучению принято говорить, что алгоритм A лучше алгоритма B, если верхняя граница 95-процентного доверительного интервала для ошибки алгоритма A меньше нижней границы 95-процентного доверительного интервала для ошибки алгоритма B.

2.4. Поиск компромисса между смещением и дисперсией для минимизации среднеквадратической ошибки

Что, если имеются две оценки, у одной из которых больше смещение, а у другой дисперсия? Какую выбрать?

Самый распространенный подход к выбору компромиссного решения – воспользоваться *перекрестной проверкой*. Эмпирически продемонстрировано, что перекрестная проверка дает отличные результаты во многих реальных задачах.

Можно также сравнить среднекваратическую ошибку (MSE) обеих оценок

$$\text{MSE} = \mathbb{E}[(\hat{\theta}_m - \theta)^2] = \text{bias}(\hat{\theta}_m)^2 + \text{Var}(\hat{\theta}_m)$$

Желательной является оценка с малой MSE, именно такие оценки держат под контролем и смещение, и дисперсию. Соотношение между смещением и дисперсией тесно связано с возникающими в машинном обучении понятиями емкости модели, недообучения и переобучения.

Если ошибка обобщения измеряется посредством MSE (и тогда смещение и дисперсия становятся важными компонентами ошибки обобщения), то увеличение емкости (то есть *усложение модели*) влечет за собой *повышение дисперсии* и *снижение смещения*.

2.5. Состоятельность

Обычно нас интересует также поведение оценки по мере роста размера обучающего набора. В частности, мы хотим, чтобы при увеличении числа примеров точечные оценки сходились к истинным значениям соответствующих параметров.

Формально это записывается в виде (условие состоятельности)

$$\hat{\theta}_m \xrightarrow{\mathbf{P}} \theta, \ (m \to \infty)$$

Иногда это условие называют *слабой состоятельностью*, понимая под *сильной состоятельностью* сходимость *почти наверное* $\hat{\theta}$ к θ.

Состоятельность гарантирует, что смещение оценки уменьшается с ростом числа примеров. Однако обратное неверно – из асимптотической несмещенности не вытекает состоятельность. Рассмотрим, к примеру, оценивание среднего μ нормального распределения $N(x; \mu, \sigma^2)$ по набору данных, содержащему m примеров: $\{x^{(1)}, \ldots, x^{(m)}\}$.

Можно было бы взять в качестве оценки первый пример: $\hat{\theta} = x^{(i)}$. В таком случае $\mathbb{E}(\hat{\theta})_m = \theta$, поэтому оценка является несмещенной вне зависимости от того, сколько примеров мы видели. Отсюда, конечно, следует, что оценка асимптотически несмещенная. Но она не является состоятельной, т.к. *неверно*, что $\hat{\theta}_m \to \theta, \ (m \to \infty)$.

2.6. Оценка максимального праводподобия

Рассмотрим множества m примеров $\mathbb{X} = \{x^{(1)}, \ldots, x^{(m)}\}$, независимо выбираемых из неизвестного порождающего распределения $p_{data}(x)$.

Обозначим $p_{model}(x; \theta)$ параметрическое семейство распределений вероятности над одним и тем же пространством, индексированное параметром θ.

Тогда оценка максимального правдоподобия для θ определяется формулой

$$\theta_{ML} = \arg\max_{\theta} p_{model}(\mathbb{X};\theta) = \arg\max_{\theta} \prod_{i=1}^{m} p_{model}(x^{(i)};\theta)$$

Такое произведение большого числа вероятностей по ряду причин может быть неудобно. Например, оно подвержено *потере значимости*. Для получения эквивалентной, но более удобной задачи оптимизации заметим, что взятие логарифма правдоподобия не изменяет arg max, но преобразует произведение в сумму

$$\theta_{ML} = \arg\max_{\theta} \sum_{i=1}^{m} \log p_{model}(x^{(i)};\theta)$$

Поскольку arg max не изменяется при умножении функции стоимости на константу, мы можем разделить правую часть на m и получить выражение в виде математического ожидания относительно эмпирического распределения $\hat{p}_{data}$, определяемого обучающими данными

$$\theta_{ML} = \arg\max_{\theta} \mathbb{E}_{x\sim\hat{p}_{data}}[\log p_{model}(x;\theta)]$$

Один из способов интерпретации оценки максимального правдоподобия состоит в том, чтобы рассматривать ее как минимизацию дивергенции (расхождения) Кульбака-Лейблера между этими эмпирическим распределением $\hat{p}_{data}$, определяемым обучающим набором, и модельным распределением.

Дивергенция Кульбака-Лейблера определяется формулой

$$D_{KL}(\hat{p}_{data}\,||\,p_{model}) = \mathbb{E}_{x\sim\hat{p}_{data}}[\log\hat{p}_{data}(x) - \log p_{model}(x)]$$

Первый член разности в квадратных скобках зависит только от порождающего данные процесса, но не от модели. Следовательно, при обучении модели, минимизирующей дивергенцию КЛ, мы должны минимизировать только величину

$$-\mathbb{E}_{x\sim\hat{p}_{data}}[\log p_{model}(x)],$$

а это, конечно, то же самое, что максимизация величины $\theta_{ML} = \arg\max_{\theta} \mathbb{E}_{x\sim\hat{p}_{data}}[\log p_{model}(x;\theta)]$.

NB: То есть, другими словами задача максимизации правдоподобия эквивалентна задаче минимизации дивергенции Кульбака-Лейблера между эмпирическим распределением $\hat{p}_{data}$ и модельным распределением p_{model}.

2.7. Метод опорных векторов

Линейную функцию в методе опорных векторов можно переписать в виде

$$w^T x + b = b + \sum_{i=1}^{m} \alpha_i x^T x^{(i)},$$

где $x^{(i)}$ – обучающий пример, α – вектор коэффициентов.

Записав алгоритм обучения в таком виде, мы сможем заменить x результатом заданной функции признаков $\varphi(x)$, а скалярное произведение – функцией $k(x, x^{(i)}) = \varphi(x) \cdot \varphi(x^{(i)})$, которая называется ядром.

Заменив скалярное произведение вычислением ядра, мы можем делать предсказание, пользуясь функцией

$$f(x) = b + \sum_i \alpha_i k(x, x^{(i)})$$

Основанная на ядре функция в точности эквивалентна предварительной обработке путем применения $\varphi(x)$ ко всем входным данным с последующим обучением линейной модели в новом преобразованном пространстве.

NB: Трюк с ядром полезен по двум причинам:

- Во-первых, он позволяет обучать модели, *нелинейно* зависящие от x, применяя методы выпуклой оптимизации, о которых точно известно, что они сходятся эффективно
- Во-вторых, *ядерная функция* k часто допускает реализацию, значительно *более эффективную с вычислительной точки зрения*, чем наивное построение двух векторов $\varphi(x)$ и явное вычисление их скалярного произведения

Главный недостаток ядерных методов – тот факт, что сложность вычисления решающей функции линейно зависит от числа обучающих примеров, поскольку i-ый пример вносит член $\alpha_i k(x, x^{(i)})$ в решающую функцию.

В методе опорных векторов эта проблема сглаживается тем, что обучаемый вектор α содержит в основном нули. Тогда для классификации нового примера *требуется вычислить ядерную функцию только для обучающих примеров с ненулевыми* α_i. Эти обучающие примеры и называются опорными векторами.

2.8. Метод главных компонент

Метод главных компонент находит ортогональное линейное преобразование, переводящее входные данные x в представление z.

Рассмотрим матрицу плана X размера $m \times n$. Будем предполагать, что матемаческое ожидание данных $\mathbb{E}[x] = 0$. Если это не так, центрирования легко добиться, вычтя среднее из всех примеров на этапе предварительной обработки.

Несмещенная выборочная ковариационная матрица, ассоциированная с X, определяется по формуле

$$Var[x] = \frac{1}{m-1} X^T X$$

PCA находит представление (посредством линейного преобразования) $z = W^T x$, для которого $Var[z]$ – *диагональная.*

Главные компоненты можно получить с помощью сингулярного разложения. Точнее, это правые сингулярные верткторы. Чтобы убедиться в этом, предположим, что W – правые сингулярные векторы в разложении $X = U\Sigma W^T$. Тогда исходное уравнение собственных векторов можно переписать в базисе W

$$X^T X = (U\Sigma W^T)^T U\Sigma W^T = W\Sigma^2 W^T$$

Разложение SVD полезно для доказательства того, что PCA приводит к диагональной матрице $Var[z]$. Применяя сингулярное разложение X, мы можем выразить дисперсию X в виде

$$Var[x] = \frac{1}{m-1}X^TX = \frac{1}{m-1}(U\Sigma^2W^T)^TU\Sigma W^T = \frac{1}{m-1}W\Sigma^2W^T,$$

где используется тот факт, что $U^TU = I$, поскольку матрца U в сингулярном разложении по определению ортогональная. Отсюда следует, что ковариационная матрица z диагональная

$$Var[z] = \frac{1}{m-1}Z^TZ = \frac{1}{m-1}W^TX^TXW = \frac{1}{m-1}W^TW\Sigma^2W^TW = \frac{1}{m-1}\Sigma^2$$

На этот раз мы воспользовались тем, что $W^TW = I$ – опять же по определению сингулярного разложения.

Проведенный анализ показывает, что представление, полученное в результате проецирования данных x на z посредством линейного преобразования W, имеет *диагональную ковариационную матрицу* Σ^2. А отсюда сразу вытекает, что *взаимная корреляция отдельных элементов* z *равна нулю.*

2.9. Стохастический градиентный спуск

Стохастический градиентный спуск (СГС) имеет важные применения и за пределами глубокого обучения. Это основной способ обучения *больших линейных моделей* на очень больших наборах данных. Для модели фиксированноо размера стоимость одного шага СГС не зависит от размера обучающего набора m. Количество шагов до достижения сходимости обычно возрастает с ростом размера обучающего набора. Но когда m стремится к бесконечности, модель в итоге сходится к наилучшей возможной ошибке тестирования еще до того, как СГС проверил каждый пример из обучающего набора. Дальнейшее увеличение m не увеличивает время обуччения, необходимое для достижения наилучшей ошибки тестирования. С этой точки зрения можно сказать, что асимптотическая стоимость обучения модели методом СГС как функции от m имеет порядок $O(1)$.

2.10. Условное логарифмическое правдоподобие и среднеквадратическая ошибка

Линейую регрессию можно интерпретировать как *нахождение оценки максимального правдоподобия*. Будем считать, что цель не в том, чтобы вернуть одно предсказание $\hat{y}$, а чтобы построить модель, порождающую условное распределение $p(y|\mathbf{x})$. Цель алгоритма обучения теперь – аппроксимировать $p(y|\mathbf{x})$, подогнав его под все эти разные значения y, совместимые с $\mathbf{x}$.

Для вывода такого же алгоритма линейной регрессии, как и раньше, определим

$$p(y|\mathbf{x}) = N(y; \hat{y}(\mathbf{x}, \mathbf{w}), \sigma^2)$$

Функкция $\hat{y}(\mathbf{x}; \mathbf{w})$ дает предсказание среднего значения нормального распределения. В этом примере мы предполагаем, что дисперсия фиксирована и равна константе σ^2. Поскольку предполагается, что примеры независимы и одинаково распределены, то условное логарифмическое

правдоподобие записывается в виде

$$\sum_{i=1}^{m} \log p(y^{(i)}|\mathbf{x}^{(i)};\theta) = -m\,\log\sigma - \frac{m}{2}\log 2\pi - \sum_{i=1}^{m}\frac{\|\hat{y}^{(i)} - y^{(i)}\|^2}{2\sigma^2},$$

где $\hat{y}^{(i)}$ – результат линейной регрессии для i-ого примера $\mathbf{x}^{(i)}$, а m – число обучающих примеров.

Сравнивая логарифмическое правдоподобие со среднеквадратической ошибкой

$$\mathrm{MSE}_{train} = \frac{1}{m}\sum_{i=1}^{m}\|\hat{y}^{(i)} - y^{(i)}\|^2,$$

мы сразу же видим, что *максимизация логарифмического правдоподобия* относительно $\mathbf{w}$ дает ту же оценку параметров $\mathbf{w}$, что *минимизация среднеквадратической ошибки*.

Значения этих критериев различны, но положение оптимума совпадает. Это служит обоснованием использования среднеквадратической ошибки в качестве оценки максимального правдоподобия.

3. Глубокие сети прямого распространения

3.1. Обучение условных распределений с помощью максимального правдоподобия

Большинство современных нейронных сетей обучается *с помощью максимального правдоподобия*. Это означает, что в качестве функции стоимости берется отрицательное логарифмическое правдоподобие, которое можно эквивалентно описать как перекрестную энтропию между обучающими данными и распределением модели

$$J(\theta) = -\mathbb{E}_{x,y\sim\hat{p}_{\mathrm{data}}}\log p_{\mathrm{model}}(\mathbf{y}|\mathbf{x})$$

Одно необычное свойство перекрестной энтропии, используемой при вычислении оценки максимального правдоподобия, заключается в том, что для типичных встречающихся на практике моделей у нее, как правило, нет минимального значения. Если выходная величина дискретна, то в большинстве моделей параметризация устроена так, что модель неспособна представить вероятность 0 или 1, но может подойти к ней сколь угодно близко. Примером может служить логистическая регрессия.

3.2. Сигмоидные блоки и выходное распределение Бернулли

Во многих задачах требуется предсказывать значение бинарной величины y. В таком виде можно представить задачу классификации с двумя классами.

Подход на основе максимального правдоподобия заключается в определении распределения Бернулли величины y при условии $\mathbf{x}$.

Градиент **0** обычно приводит к проблемам, потому что у алгоритма обучения нет никаких указаний на то, как улучшить параметры.

Лучше применять другой подход, который гарантирует, что градиент обязательно будет достаточно большим, если модель дает неверный ответ. Этот подход основан на использовании сигмоидных выходных блоков в сочетании с максимальным правдоподобием.

Сигмоидный выходной блок

$$\hat{y} = \sigma(\mathbf{w}^T\mathbf{h} + b),$$

где σ – логистическая сигмоида.

3.3. Блоки softmax и категориальное выходное распределение

Если требуется представить распределение вероятности дискретной случайной величины, принимающей n значений, то можно воспользоваться функцией softmax. Ее можно рассматривать как обобщение сигмоиды, которая использовалась для представления распределения бинарной величины.

Функция softmax чаще всего используется как выход классификатора для представления распределения вероятности n классов. Реже функция softmax используется внутри самой модели.

Как и сигмоида, функция активации softmax склонна к насыщению. У сигмоиды всего один выход, и она насыщается, когда абсолютная величина аргумента очень велика. У softmax выходных значений несколько. Они насыщаются, когда велика абсолютная величина разностей между входными значениями.

softmax – это сглаженный вариант $\arg\max$. Сложность спектрального разложения матрицы $d \times d$ имеет порядок $O(d^3)$.

3.4. Скрытые блоки

Некоторые скрытые блоки, не являются всюду дифференцируемыми. Например, функция линейной ректификации (ReLU) $g(z) = \max\{0, z\}$ не дифференцируема в точке $z = 0$. Может показаться, что из-за этого g непригодна для работы с алгоритмами обучения градиентными методами. Но на практике градиентный спуск работает для таких моделей машинного обучения достаточно хорошо. Отчасти это связано с тем, что алгоритмы обучения нейронных сетей обычно не достигают локального минимума функции стоимости, а просто находят достаточно малое значение. Поскольку мы не ожидаем, что обучение выйдет на точку, где градиент равен 0, то можно смириться с тем, что минимум функции стоимости соответствует точкам, в которых градиент не определен. Недифференцируемые скрытые блоки обычно не дифференцируемы лишь в немногих точках. В общем случае функцию $g(z)$ имеет производную слева, определяемую коэффициентом наклона функции слева от z, и аналогично производную справа. Функция дифференцируема в точке z, только если производные слева и справа определены и равным между собой. Для функции $g(z) = \max\{0, z\}$ производная слева в точке $z = 0$ равна 0, а производная справа равна 1. В программных реализациях обучения нейронной сети обычно возвращается какая-то односторонняя производная, а не сообщается, что производная не определена и не возбуждается исключение. Эвристически это можно оправдать, заметив, что градиентная оптимизация на цифровом компьютере в любом случае подвержена численным погрешностям. Когда мы просим вычислить $g(0)$, крайне маловероятно, что истинное значение действительно равно 0. Скорее всего, это какое-то малое значение, округленное до 0.

Важно, что на практике можно *спокойно игнорировать недифференцируемость функции активации* скрытых блоков.

3.4.1. Блоки линейной ректификации и их обобщения

В блоке линейной ректификации используется функция активации $g(z) = \max\{0, z\}$. Эти блоки легко оптимизировать, потому что они очень похожи на линейные. Разница только в том, что блок линейной ректификации в половине своей области определения выводит 0. Поэтому производная блока линейной ректификации остается большой всюду, *где блок активен*.

Блоки линейной ректификации обычно применяются *после* аффинного преобразования

$$h = g(W^T x + b)$$

При инициализации параметров аффинного преобразования рекомендуется присваивать всем элементам b небольшое положительное значение, например, 0.1. Тогда блок линейной ректификации в начальный момент с большой вероятностью окажется активен для большинства обучающих примеров, и производная будет отлична от нуля.

Недостатком блоков линейной ректификации является невозможность обучить их градиентными методами на примерах, для которых функция активации блока равна нулю. Различные обобщения гарантируют, что градиент имеется в любой точке.

Три обобщения блоков линейной ректификации основаны на использовании ненулевого углового коэффициента α_i, когда $z_i < 0 : h_i = \max(0, z_i) + \alpha_i \min(0, z_i)$.

В случае абсолютной ректификации берутся фиксированные значения $\alpha_i = -1$, так что $g(z) = |z|$. Такая функция активации используется при распознавании объектов в изображении, где имеет смысл искать признаки, инвариантные относительно изменения полярности освещения. Другие обобщения находят более широкие применения. В случае ReLU с утечкой α_i принимаются равными фиксированному малому значению, например, 0.01, а в случае параметрического ReLU α_i считается обучаемым параметром.

Блоки линейной ректификации и все их обобщения основаны на принципе, согласно которому модель проще обучить, если ее поведение близко к линейному.

3.4.2. Логистическая сигмоида и гиперболический тангенс

Блоки линейной ректификации стали использоваться сравнительно недавно, а раньше большинство нейронных сетей в роли функции активации применялась логистическая сигмоида или гиперболический тангенс.

Сигмоидные блоки в качестве выходных предсказывают вероятность того, что бинарная величина равна 1. В отличие от кусочно-линейных, сигмоидные блоки близки к асимптоте в большей части своей области определения – приближаются к высокому значению, когда z стремится к бесконечности, и к низкому, когда z стремится к минус бесконечности. Высокой чувствительностью они обладают только в окрестности нуля. Из-за *насыщения* сигмоидальных блоков *градиентное обучение сильно затруднено*. Поэтому использование их в качестве скрытых блоков в сетях прямого распространения ныне не рекомендуется.

Если использовать сигмоидальную функцию активации необходимо, то лучше взять не логистическую сигмоиду, а гиперболический тангенс. Он ближе к тождественной функции в том смысле, что $\tanh(0) = 0$, тогда как $\sigma(0) = 1/2$.

Поскольку tanh походит на тождественную функцию в окрестности нуля, обучение глубокой нейронной сети $\hat{y} = w^T \tanh(U^T \tanh(V^T x))$ напоминает обучение линейной модели $\hat{y} = w^T U^T V^T x$,

при условии, что сигналы активации сети удается удерживать на низком уровне. При этом обучение сети с функцией активации tanh упрощается.

Сигмоидальные функции активации все же применяются, но не в сетях прямого распространения. К реккурнетным сетям, многим вероятностным моделям и некоторым автокодировщикам предъявляются дополнительные требования, исключающие использование кусочно-линейных функций.

3.5. Правило дифференцирования сложной функции

Это правило применяется для вычисления производных функций, являющихся композициями других функций, чьи производные известны.

Пусть $\mathbf{x} \in \mathbb{R}^m, \mathbf{y} \in \mathbb{R}^n$, g отображает $\mathbb{R}^m$ в $\mathbb{R}^n$, а f отображает $\mathbb{R}^n$ в $\mathbb{R}$. Тогда правило дифференцирования сложной функции в векторной форме запишется в виде

$$\nabla_x z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_y z,$$

где $\partial \mathbf{y}/\partial \mathbf{x}$ – *матрица Якоби* функции g размера $n \times m$.

Отсюда видно, что градиент по переменной $\mathbf{x}$ можно получить, умножив матрицу Якоби $\partial \mathbf{y}/\partial \mathbf{x}$ на градиент $\nabla_y z$.

Обычно алгоритм обратного распространения применяется к тензорам произвольной размерности, а не просто к векторам. Концептуально это то же самое, что применение к векторам. Разница только в том, как числа организуются в сетке для представления тензора. Можно вообразить, что тензор сериализуется в вектор перед обратным распространением, затем вычисляется векторозначный градиент, после чего градиент снова преобразуется в тензор.

В таком переупорядоченном представлении обратное распространение – это все то же умножение якобиана на градиенты.

Для обозначения градиента значения z относительно тензора $\mathbf{X}$ мы пишем $\nabla_{\mathbf{X}} z$, как если бы $\mathbf{X}$ был просто вектором. Теперь индексы элементов $\mathbf{X}$ составные – например, трехмерный тензор индексируется тремя координатами. Мы можем абстрагироваться от этого различия, считая, что одна переменная i представляет целый кортеж индексов. Для любого возможного индексного кортежа i $(\nabla_{\mathbf{X}} z)_i$ обозначает частную производную $\partial z/\partial \mathbf{X}_i$ – точно так же, как для любого целого индекса i $(\nabla_x z)_i$ обозначает $\partial z/\partial x_i$. В этих обозначениях можно записать правило дифференцирования сложной функции в применении к тензорам. Если $\mathbf{Y} = g(\mathbf{X})$ и $z = f(\mathbf{Y})$, то

$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} Y_j) \frac{\partial z}{\partial Y_j}$$

Алгоритм обратного распространения был разработан, чтобы избежать многократного вычисления одного и того же выражения при дифференцировании сложной функции. Из-за таких повторов время выполнения наивного алгоритма могло расти экспоненциально. Теперь, можно оценить вычислительную сложность алгоритма обратного распространения.

Если предположить, что стоимость вычисления всех операций приблизительно одинакова, то вычислительную сложность можно проанализировать в терминах количества выполненных операций. Следует помнить, что под единицей мы понимаем базовую единицу графа вычислений, в действительности она может состоять из нескольких арифметических операций (например, умножение матриц может считаться одной операцией в графе). Вычисление градиента в графе

с n вершинами никогда не приводит к выполнению или сохранению результатов более $O(n^2)$. Здесь мы подсчитываем в графе вычислений, а не отдельные аппаратные операции, поэтому важно понимать, что время выполнения разных операций может значительно различаться.

Легко видеть, что для вычисления градиента требуется не более $O(n^2)$ операций, потому что на этапе прямого распространения в худшем случае будут обсчитаны все n вершин исходного графа (в зависимости от того, какие значения мы хотим вычислить, может потребоваться обойти весь граф).

Поскольку граф вычислений – это ориентированный ациклический граф, число ребер в нем не более $O(n^2)$. Для типичных графов, встречающихся на практике, ситуация даже лучше. В большинстве нейронных сетей функции стоимости имеют в основном цепную структуру, так что сложность обратного распространения равна $O(n)$. Это намного лучше, чем наивный подход, при котором число обрабатываемых вершин иногда растет *экспоненциально*!

Откуда возникает экспоненциальный рост, можно понять, раскрыв и переписав правило дифференцирования сложной функции без рекурсии (здесь сумма по путям $u^{(\pi_1, u^{(\pi_2)}), \ldots, u^{(\pi_t)}}$ из $\pi_1 = j$ в $\pi_t = n$)

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum \prod_{k=2}^{t} \frac{\partial u^{(\pi_k)}}{\partial u^{(\pi_{k-1})}}$$

Поскольку количество путей из вершины j в вершину n может экспоненциально зависит от длины пути, то число слагаемых в этой сумме, равное числу таких путей, может расти экспоненциально с увеличением глубины графа прямого распространения. Такая высокая сложность связана с многократным вычислением $\partial u^{(i)}/\partial u^{(j)}$. Чтобы *избежать повторных вычислений*, мы можем рассматривать *обратное распространение как алгоритм заполнения таблицы*, в которой храняться промежуточные результаты $\partial u^{(n)}/\partial u^{(i)}$.

Каждой вершине графа соответствует элемент таблицы, в котором хранится градиент для этой вершины. Заполняя таблицу в определенном порядке, алгоритм обратного распространения избегает повторного вычисления многих ошибок подвыражений. Такую стратегию заполнения таблицы иногда называют динамическим программированием.

3.6. Регуляризация в глубоком обучении

3.6.1. Штрафы по норме параметров

Многие подходы к регляризации основаны на ограничении емкости моделей путем прибавления штрафа по норме параметра $\Omega(\theta)$ к целевой функции J

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta),$$

где $\alpha \in [0, \infty]$ – гиперпараметр, задающий вес члена Ω, штрафующего по норме, относительно стандратной целевой функции J. Чем больше значение α, тем сильнее регуляризация.

В нейронных сетях мы обычно предпочитаем штрафовать по норме *только веса* аффинного преобразования в каждом слое, оставляя смещения нерегуляризованными.

Регуляризация параметров смещения может стать причиной значительного недообучения.

Регуляризация парамтеров по норме L_2 Полная целевая функция с регуляризацияей Тихонова имеет вид

$$\tilde{J}(w; X, y) = \frac{\alpha}{2} w^T w + J(w; X, y),$$

а градиент по параметрам

$$\nabla_w \tilde{J}(w; X, y) = \alpha w + \nabla_w J(w; X, y)$$

Один шаг обновления весов с целью уменьшения градиента имеет вид

$$w \leftarrow w - \varepsilon \left(\alpha w + \nabla_w J(w; X, y)\right)$$

То же самое можно переписать в виде

$$w \leftarrow (1 - \varepsilon\alpha) w - \varepsilon \nabla_w J(w; X, y)$$

Как видим, добавление члена сложения весов изменило правило обучения: теперь мы на каждом шаге умножаем вектор весов на постоянный коэффициент, меньший 1, перед тем как выполнить стандартное обновление градиента.

Еще упростим анализ, предположив квадратичную аппроксимацию целевой функции в окрестности того значения весов, при котором достигается минимальная стоимость обучения без регуляризации. Если целевая функция действительно квадратичная, как в случае модели линейной регрессии со среднеквадратической ошибкой, то такая аппроксимация идеальна. Аппроксимация $\tilde{J}$ описывается формулой

$$\hat{J}(\theta) = J(w^*) + 1/2(w - w^*)^T H (w - w^*),$$

где H – матрица Гессе J относительно w, вычисленная в точке w^*. В этой квадратичной аппроксимации нет члена первого порядка, потому что w^*, по определению, точка минимума, в которой градиент обращается в нуль.

Минимум $\hat{J}$ достигается там, где градиент

$$\nabla_w \hat{J}(w) = H(w - w^*) = 0$$

Чтобы изучить эффект снижения весов, прибавим градиент снижения весов. Теперь мы можем найти из него минимум регуляризованного варианта $\hat{J}$. Обозначим $\tilde{w}$ положение точки минимума.

$$\begin{aligned} \alpha\tilde{w} + H(\tilde{w} - w^*) &= 0 \\ (H + \alpha I)\tilde{w} &= Hw^* \\ \tilde{w} &= (H + \alpha I)^{-1} Hw^* \end{aligned}$$

Поскольку матрица H вещественная и симметричная, мы можем разложить ее в произведение диагональной матрицы Λ и ортогональной матрицы собственных векторов Q

$$H = Q\Lambda Q^T$$

Тогда

$$\tilde{w} = (Q\Lambda Q^T + \alpha I)^{-1} Q\Lambda Q^T w^* = Q(\Lambda + \alpha I)^{-1}\Lambda Q^T w^*.$$

Компонента w^*, параллельная i-ому собственному вектору H, умножается на коэффициент $\lambda_i/(\lambda_i + \alpha)$. Вдоль направлений, для которых собственные значения H относительно велики, например, когда $\lambda_i \gg \alpha$, эффект регуляризации сравнительно мал. Те же компоненты, для которых $\lambda_i \ll \alpha$, сжимаются почти до нуля.

Если направление не дает вклада в уменьшение целевой функции, то собственное значение гессиана мало, т.е. движение в этом направялении не приводит к заметному возрастнию градиента. Компоненты вектора весов, соответствующие таким малозначным направлениям, снижаются почти до нуля благодаря использованию регуляризации в ходе обучения.

Рассмотрим линейную регрессию, в которой истинная функция стоимости квадратическая. Для линейной регрессии функция стоимости равна сумме квадратов ошибок

$$(xw - y)^T (Xw - y)$$

После добавления L_2-регуляризации целевая функция принимает вид

$$(Xw - y)^T (Xw - y) + \frac{1}{2} w^T w$$

В результате *нормальные уравнения*, из которых ищется решение

$$w = (X^T X)^{-1} X^T y$$

принимают вид

$$w = (X^T X + \alpha I)^{-1} X^T y$$

Матрица $X^T X$ пропорциональна *ковариационной матрице* $1/m X^T X$. Применение L_2-регуляризации заменяет эту матрицу на $(X^T X + \alpha I)^{-1}$. Новая матрица отличается от исходной только прибавлением α ко всем диагональным элементам. *Диагональные элементы* этой матрицы соответствуют *дисперсии каждого входного признака.*

Таким образом, *L_2-регуляризация* заставляет алгоритмы обучения «воспринимать» вход X *как имеющий более высокую дисперсию* и, следовательно, уменьшать веса тех признаков, для которых ковариация с выходными метками мала, по сравнению с добавленной дисперсией.

NB: L_2-регуляризация занижает веса признаков, которые обнаруживают слабую ковариацию с целевой меткой.

L_1-регуляризация Формально L_1-регуляризация параметров модели w определяется по формуле

$$\Omega(\theta) = \|w\|_1 = \sum_i |w_i|,$$

т.е. как сумма абсолютных величин отдельных параметров.

Регуляризованная целевая функция описывается формулой

$$\tilde{J}(w; X, y) = \alpha\|w\|_1 + J(w; X, y),$$

а ее градиент (точнее, *частичный градиент*) равен

$$\nabla_w \tilde{J}(w; X, y) = \alpha \operatorname{sign}(w) + \nabla_w J(w; X, y),$$

где $\operatorname{sign}(w)$ означает, что функция sign применяется к каждому элементу w.

Теперь вклад регуляризации в градиент уже не масштабируется линейно с ростом каждого w_i, а описывается постоянным слагаемым, знак которого совпадает с $\operatorname{sign}(w_i)$. Один из следствий является тот факт, что мы уже не получим изящных алгебраических выражений квадратичной аппроксимации $J(X; y, w)$, как в случае L_2-регуляризации.

Квадратичную аппроксимацию L_1-регуляризованной целевой функции можно представить в виде суммы по параметрам

$$\hat{J}(w; X, y) = J(w; X, y) + \sum_i \Big[\frac{1}{2} H_{i,i}(w_i - w_i^*)^2 + \alpha|w_i| \Big]$$

У задачи минимизации этой приближенной функции стоимости имеется аналитическое решение (для каждого измерения i) вида

$$w_i = \operatorname{sign}(w_i^*) \max \Big[|w_i^*| - \frac{\alpha}{H_{i,i}}, 0 \Big]$$

Предполагается, что матрица Гессе диагональная $H = \operatorname{diag}([H_{1,1}, \ldots, H_{n,n}])$, где все $H_{i,i} > 0$.

Предположим, что $w_i^* > 0$ для всех i. Тогда есть два случая:

1. $w_i^* \leqslant \alpha / H_{i,i}$. Тогда оптимальное значение w_i для регуляризованной целевой функции будет просто $w_i = 0$,
2. $w_i^* > \dfrac{\alpha}{H_{i,i}}$. Тогда регуляризация не сдвигает оптимальное значение w_i в нуль, а просто смещает его в этом направлении на расстояние $\alpha / H_{i,i}$.

Аналогичное рассуждение проходит, когда $w_i^* < 0$, только L_1-штраф увеличивает w_i на $\alpha / H_{i,i}$ или обращает в 0.

По сравнению с L_2-регуляризацией, L_1-регуляризация дает более *разреженное* решение. В этом контексте под разреженностью понимается тот факт, что у некоторых параметров оптимальное значение равно 0.

Свойство разреженности, присущее L_1-регуляризации, активно эксплуатировалось как механизм отбора признаков, идея которого состоит в том, чтобы упростить задачу машинного обучения за счет выбора некоторого подмножества располагаемых признаков. В частности, хорошо известная модель LASSO (Least Absolute Shrinkage and Selection Operator) объединяет L_1-штраф

с линейной моделью и среднеквадратической функцией стоимости. Благодаря L_1-штрафу некоторые веса обращаются в 0, и соответствующие им признаки отбрасываются.

Многие *стратегии регуляризации* можно интерпретировать как *байесовский вывод* на основе *оценки апостериорного максимума* (MAP). В частности, L_2-регуляризация эквивалента байесовскому выводу на основе MAP с *априорным нормальным распределением весов*. В случае L_1-регуляризации штраф $\alpha\Omega(w) = \alpha \sum_i |w_i|$, применяемый для регуляризации функции стоимости, эквивалентен члену, содержащему логарифм априорного расределения, который максимизируется байесовским выводом на основе MAP, когда в качестве *априорного* используется *изотропное распределение Лапласа векторов* $w \in \mathbb{R}^n$.

3.7. Регуляризация и неопределенные задачи

В некоторых случаях без регуляризации просто невозможно корректно поставить задачу машинного обучения. Многие линейные модели в машинном обучении, в т.ч. линейная регрессия, включают обращение матрицы $X^T X$. *Это невозможно, если* $X^T X$ *сингулярная.*

Матрица может оказаться сингулярной, если порождающее распределение действительно не имеет дисперсии в некотором направлении, или если в некотором направении не наблюдается дисперсии, потому что число примеров (строк X) меньше числа входных признаков (столбцов X). В таком случае во многих вариантах регуляризации прибегают к обращению матрицы $X^T X + \alpha I$. Гарантируется, что такая регуляризованная матрица обратима.

NB: регуляризованная матрица $(X^T X + \alpha I)$ обратима всегда!

3.8. Ранняя остановка

Каким образом ранняя остановка выступает в роли регуляризатора. Мы уже не раз отмечали, что ранняя остановка ялвяется стратегией регуляризации, но в обоснование этого заявления привели только кривые обучения, на которых ошибка на контрольном наборе имеет U-образную форму. А каков истинный механизм регуляризации модели с помощью ранней остановки?

Допустим, что выбрано τ шагов оптимизации (что соответствует τ итерациям обучения) и скорость обучения ε. Произведение $\varepsilon \cdot \tau$ можно рассматривать как *меру эффективной емкости.* В предположении, что градиент ограничен, наложение ограничений на число итераций и скорость обучения лимитируют область пространства параметров, достижимую из θ_0. В этом смысле $\varepsilon \cdot \tau$ ведет себя как величина, обратная коэффициенту снижения весов.

То есть чем больше итераций отводится для обучения или чем выше скорость обучения, тем сложнее модель и потому тем слабее регуляризация.

Значения параметров, соответствующие направялениям *сильной кривизны* целевой функции, *регуляризируются меньше*, чем в направлениях меньшей кривизны.

Конечно, ранняя остановка – больше, чем простое ограничение на длину траектории; *ранняя остановка* обычно подразумевает наблюдение за ошибкой на *контрольном наборе*, чтобы оборвать траекторию в удачной точке пространства.

Поэтому, по сравнению со снижением весов, у ранней остановки есть то преимущество, что она автоматически определяет правильную степень регуляризации, тогда как при использовании снижения весов требуется много экспериментов с разными значениями-гиперпараметрами.

3.9. Баггинг и другие ансамблевые методы

Баггинг – метод умеьшения ошибки обобщения путем комбинирования нескольких моделей. Идея заключается в том, чтобы раздельно обучить разные модели, а затем организовать их голосования за результаты на тестовых примерах. Это частный случай общей стратегии машинного обучения – усреднения моделей. Методы, в которых эта стратегия используется, называется ансамблевыми методами.

В качестве примера рассмотрим набор из k моделей регрессии. Предположим, что каждая модель делает ошибку ε_i на каждом примере, причем ошибки имеют многомерное нормальное распределение с нулевым средним, дисперсиями $\mathbb{E}[\varepsilon_i^2] = v$ и ковариациями $\mathbb{E}[v_i v_j] = c$. Тогда ошибка, полученная в результате усреднения предсказаний всего ансамбля моделей, равна $(1/k)\sum_i \varepsilon_i$.

Математическое ожидание квадрата ошибки ансамблевого предиктора равно

$$\mathbb{E}\Big[\big(\frac{1}{k}\sum_i \varepsilon_i\big)^2\Big] = \frac{1}{k^2}\mathbb{E}\Big[\sum_i \big(\varepsilon_i^2 + \sum_{j\neq i}\varepsilon_i\varepsilon_j\big)\Big] = \frac{1}{k}v + \frac{k-1}{k}c.$$

В случае, когда ошибки идеально коррелированы, т.е. $c = v$, среднеквадратическая ошибка сводится к v, так что усреднение моделей ничем не помогает. В случае, когда ошибки вообще некоррелированы, т.е. $c = 0$, среднеквадратическая ошибка ансамбля равна всего $(1/k)v$ и, значет, линейно убывает с ростом размера ансамбля. Иными словами, в среднем показывает качество не хуже любого из его членов, а если члены совершают независимые ошибки, то ансамбль работает значительно лучше своих членов.

Точнее, баггинг подразумевает построение k разных наборов данных. В каждом наборе столько же примеров, сколько в исходном, но строятся они путем *выборки с возвращением* из исходного набора. Это означает, что с высокой вероятностью в каждом наборе данных отуствуют некоторые примеры из исходного набора и присутствуют несколько дубликатов (в среднем, если размер результирующего набора равен размеру исходного, в него попадает две трети примеров из исходного набора).

Затем i-ая модель обучается на i-ом наборе данных. Различия в составе примеров, включеных в набор, обуславливают различия между обученными моделями.

Нейронные сети дают настолько широкое разнообразие решений, что усреднение моделей может оказаться выгодным, даже если все модели обучались на одном и том же наборе данных. Различия, обусловленные случайной инициализацией, случайным выбором мини-пакетов, разными гиперпараметрами и результатами недетрминированной реализации нейронной сети, зачастую достаточны, чтобы различные члены ансамбля допускали частично независимые ошибки.

Усреднение моделей – исключительно мощный и надежный метод *уменьшения ошибки обобщения.*

3.10. Прореживание

Прореживание (dropout) – это вычислительно недорогой, но мощный метод *регуляризации* широкого семейства моделей. Прореживание предлагает дешевую аппроксимацию обучения и вычисления баггингового ансамбля экспоненциально большого числа нейронных сетей.

Точнее говоря, в процессе прореживания обучается *ансамбль*, состоящий из *подсетей*, получаемых *удалением невыходных нейронов сетей.*

Напомним, что для обучения методом баггинга мы определяем k различных моделей, строим k различных наборов данных путем выборки с возвращением из обучающего набора, а затем обучаем i-ю модель на i-ом наборе.

Цель *прореживания* – аппроксимировать этот процесс на экспоненциально большом числе нейронных сетей. Точнее говоря, для обучения методом прореживания мы используем алгоритм, основанный на мини-пакетах, который делает небольшие шаги, например алгоритм стохастического градиентного спуска. При загрузке *каждого примера* в мини-пакет мы случайным образом генерируем битовую маску, применяемую ко всем *входным* и *скрытым* блокам сети. Элемент маски для каждого блока выбирается независимо от всех остальных.

Вероятность включить в маску значение 1 (означающее, что соответствующий блок включается) – гиперпараметр, фиксируемый до начала обучения, а не функция текущего значения параметров модели или входного примера. Обычно входной блок включается с вероятностью 0.8, а скрытый – с вероятностью 0.5. Затем, как обычно, производится прямое распространение, обратное распространение и обновление.

Поскольку обычно принимается вероятность включения 0.5, то правило масштабирования весов сводится к делению весов пополам в конце обучения, после чего модель используется как обычно. Как бы то ни было, цель состоит в том, чтобы ожидаемому суммарному входу в блок на этапе тестирования был приблизительно равен ожидаемому суммарному входу в тот же блок на этапе обучения, несмотря на то что в среднем половина блоков во время обучения отсутствует.

Прореживание эффективнее других стандартных вычислительно недорогих регуляризаторов: снижения весов, фильтрации с ограничениями по норме и разреженной активации. Дальнейшего улучшения можно добиться, *комбинируя* прореживание с другими видами регуляризации.

У прореживания есть еще одно важное преимущество: оно не налагает существенных ограничений на тип модели или процедуру обучения. Оно одинаково хорошо рабоатет практически с любой моделью, если в ней используется распределенное представление и ее можно обучить методом стохастического градиентного спуска.

Хотя стоимость одного шага применения прореживания к конкретной модели пренебрежимо мала, его общая стоимость для модели в целом может оказаться значительной. Будучи методом регуляризации, *прореживание уменьшает эффективную емкость модели*. Чтобы компенсировать этот эффект, мы должны увеличить размер модели. Как правило, оптимальная ошибка на контрольном наборе при использовании прореживания намного ниже, но расплачиваться за это приходится гораздо большим размером модели и числом итераций алгоритма обучения.

NB: Для очень больших наборов данных регуляризация не сильно снижает ошибку обобщения. В таких случаях вычислительная стоимость прореживания и увеличения модели могут перевесить выигрыш от регуляризации.

Прореживание регуляризует каждый скрытый блок, делая его не просто хорошим признаком, а *хорошим в разных контекстах*.

Пакетная нормализация изменяет параметризацию модели, стремясь ввести в скрытые блоки как аддитивный, так и мультипликативный шум на этапе обучения. Основная цель *пакетной нормализации* – улучшить оптимизацию, но *шум может давать регуляризующий эффект*, так что иногда прореживание оказывается лишним.

4. Оптимизация в обучении глубоких моделей

4.1. Чем обучение отличается от чистой оптимизации

В большинстве ситуаций нас интересует некоторая *мера качества* P, которая определена относительно *тестового набора* и может оказаться вычислительно неприступной. Поэтому мы оптимизируем P *косвенно*. Мы уменьшаем другую функцию стоимости $J(\theta)$ в надежде, что при этом улучшится и P. Это резко отличается от чистой оптимизации, где минимизация J и есть конечная цель.

Типичную функцию стоимости можно представить в виде среднего по *обучающему набору*

$$J(\theta) = \mathbb{E}_{(\mathbf{x},y)\sim\hat{p}_{data}} L(f(\mathbf{x};\theta), y),$$

где L – функия потерь на одном примере, $f(\mathbf{x};\theta)$ – предсказанный выход для входа $\curvearrowleft$, а $\hat{p}_{data}$ – эмпирическое распределение.

Мы обычно предпочетаем минимизировать соответствующую целевую функцию, в которой математическое ожидание берется по порождающему данные распределению p_{data}, а не просто по конечному набору

$$J^*(\theta) = \mathbb{E}_{(\mathbf{x},y)\sim p_{data}} L(f(\mathbf{x};\theta), y)$$

4.1.1. Минимизация эмпирического риска

Цель алгоритма машинного обучения – уменьшить математическое ожидание ошибки обощения. Эта величина называется *риском*. Подчеркнем, что математическое ожидание берется по истинному распределению p_{data}. Если бы мы знали истинное распределение $p_{data}(\mathbf{x}, y)$, то минимизация риска была бы задачей оптимизации, решаемой с помощью алгоритма оптимизации. Но когда $p_{data}(\mathbf{x}, y)$ неизвестно, а есть только обучающий набор примеров, мы имеем задачу машинного обучения.

Простейший способ преобразовать задачу машинного обучения в задачу оптимизации – минимизировать ожидаемые потери на обучающем наборе. Это значит, что мы заменяем истинное распределение $p(\mathbf{x}, y)$ *эмпирическим распределением* $\hat{p}(\mathbf{x}, y)$, определяемым по *обучающему набору*. И теперь требуется минимизировать *эмпирический риск*

$$\mathbb{E}_{(\mathbf{x},y)\sim\hat{p}_{data}} L(f(\mathbf{x};\theta), y) = \frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}^{(i)};\theta), y^{(i)}),$$

где m – количество *обучающих примеров*.

4.2. Суррогатные функции потерь и ранняя остановка

Иногда реальтно интересующая нас функция потерь (скажем, ошибка классификации) и та, что может быть эффективно оптимизирована, – «две большие разницы». В таких случаях оптимизируют *суррогатную функцию потерь*, выступающую в роли заместителя истинной, но обладающую рядом преимуществ.

В некоторых случаях суррогатная функция потерь позволяет достичь даже больших успехов в обучении. Например, на тестовом наборе бинарная потеря продолжает уменьшаться еще долго

после того, как на обучающем наборе достигла нуля, если обучение производилось с использованием суррогата в виде логарифмического правдоподобия. Объясняется это тем, что даже когда ожидаемая бинарная потеря равна 0, робастность классификатора можно еще улучшить, отодвинув классы дальше друг от друга и получив тем самым более уверенный и надежный классификатор, который извлекает больше информации из обучающих данных, чем было бы возможно в случае простой минимизации средней бинарной потери на обучающем наборе.

Обучение зачастую заканчивается, когда производные суррогатной функции потерь все еще велики, и этим разительно отличается от чистой оптимизации, при которой считается, что алгоритм сошолся, если градиент стал очень малым.

4.3. Пакетные и мини-пакетные алгоритмы

Еще одно отличие алгоритмов машинного обучения от общих алгоритмов оптимизации состоит в том, что целевая функция обычно представляет собой представлена в виде суммы по обучающим примерам.

Чаще всего используется градиент

$$\nabla_\theta J(\theta) = \mathbb{E}_{\mathbf{x},y\sim\hat{p}_{data}} \nabla_\theta \log p_{model}(\mathbf{x}, y; \theta)$$

Вычисление точного значения этого математического ожидания *обошлось бы очень дорого*, потому что для этого нужно вычислить модель на каждом примере из набора данных. На практике можно случайно выбрать небольшое число примеров и усреднить только по ним!

Алгоритмы оптимизации, в которых используется *весь обучающий пакет*, называются *пакетными градиентными методами*, поскольку обрабатывают сразу все примеры одним большим пакетом.

Алгоритмы оптимизации, в которых используется *по одному примеру за раз*, называют *стохастическими методами*.

Большинство алгоритмов, используемых в глубоком обучении, находится где-то посередине – число примеров в них больше одного, но меньше размера обучающего набора. Традиционно они назывались *мини-пакетными стохастическими методами*.

На размер мини-пакета оказывают влияние следующие факторы:

- чем больше пакет, тем точнее оценка градиента, но зависимость хуже линейной,
- если пакет очень мал, то не удается в полной мере задействовать преимущества многоядерной архитектуры. Поэтому существует некий абсолютный минимум размера пакета – такой, что обработка мини-пакетов меньшего размера не дает никакого выигрыша во времени,
- если все примеры из пакета нужно обрабатывать параллельно (так обычно и бывает), то размер пакета лимитирован объемом памяти,
- для некотрых видов оборудования оптимальное время выполнения достигается при определенных размерах массива. Так, для GPU наилучшие результаты получаются, когда размер пакета – степень 2. Типичный паке имеет размер от 32 до 256, а для особо больших моделей иногда пробуют 16,
- небольшие пакеты могут дать эффект регуляризации. Ошибка обобщения часто оказывается наилучшей для пакета размера 1. Но общее время работы может оказаться очень большим из-за увеличения числа шагов – как из-за пониженной скорости обучения, так и потому, что для перебора всего обучающего набора требуется больше шагов.

Методы, которые вычисляют обновления *только на основе градиента* g, обычно сравнительно устойчивы и могут работать с *пакетами небольшого размера*, порядка 100.

Методы второго порядка, в которых используется также *матрица Гессе* H и которые вычисляют такие обновления, как $H^{-1}g$, обычно нуждаются в пакетах гораздо большего размера, порядка 10 000. Такие большие пакеты нужны, чтобы свести к минимуму флуктуации в оценках $H^{-1}g$.

Важно также, чтобы мини-пакеты выбирались случайно! Для вычисления несмещенной оценки ожидаемого градиента по выборке необходимо, чтобы *примеры* были *независимы*. Мы также хотим, чтобы две последовательные оценки градиента были независимы друг от друга, поэтому *два последовательных мини-пакета примеров* тоже должны быть *независимы*.

В тех случаях, когда порядок примеров в наборе не случаен, необходимо перетасовать пакет, прежде чем формировать мини-пакеты. Для очень больших наборов, насчитывающих миллиарды примеров, выбирать примеры по-настоящему случайно при каждом построении мини-пакета не всегда возможно. К счастью, на практике обычно достаточно перетасовать набор один раз и затем хранить его в таком виде. При этом получается фиксированный набор возможных мини-пакетов последовательных примеров, которым вынуждены будут пользоваться все обучаемые впоследствие модели, и каждая модель будет видеть примеры в одном и том же порядке при проходе по обучающим данным. Но похоже, что такое отклонение от истинно случайного выбора не оказывает значимого негативного эффекта. Тогда как полное пренебрежение перетасовкой примеров способно серьезно снизить эффективность алгоритма.

Интересным обоснованием мини-пакетного стохастического градиентного спуска является тот факт, что он происходит в направлении градиента истинной ошибки обобщения, при условии что примеры не повторяются. В большинстве реализаций этого алгоритма набор данных *перетасовывается один раз*, после чего по нему производится *несколько проходов*. На первом проходе каждый мини-пакет используется для вычисления несмещенной оценки истинной ошибки обобщения. На втором проходе оценка становится смещенной, потому что получена повторной выборкой уже использованных значений, а не новых примеров из порождающего распределения.

В тех случаях, когда размер набора данных растет быстрее, чем вычислительные ресурсы для его обработки, все чаще в машинном обучении переходят к практике, когда *каждый обучающий пример используется ровно один раз*, или даже производится неполный проход по обучающему набору.

4.4. Проблемы оптимизации нейронных сетей

Традиционно в машинном обучении избегали сложностей общей оптимизации за счет тщательного выбора целевой функции и ограничений, гарантирующих выпуклость задачи оптимизации. При обучении нейронных сетей приходится сталкиваться с общим невыпуклым случаем.

4.4.1. Плохая обусловленность

Ряд проблем возникает *даже при оптимизации выпуклых функций*. Самая известная из них – *плохая обусловленность матрицы Гессе* H. Это очень общая проблема, присущая большинству методов численной оптимизации, все равно, выпуклой или нет.

Считается, что проблема плохой обусловленности присутствует во всех задачах обучения нейронных сетей. Она может проявляться в «застревании» СГС в том смысле, что *даже очень малые шаги увеличивают функцию стоимости*.

Разложении функции стоимости в ряд Тейлора до членов второго порядка показывает, что шаг гадиентного спуска величиной $-\varepsilon \mathbf{g}$ увеличивает стоимость на

$$1/2\,\varepsilon^2 \mathbf{g}^T H \mathbf{g} - \varepsilon \mathbf{g}^T \mathbf{g}$$

Плохая обусловленность градиента становится проблемой, когда $1/2\,\varepsilon^2 \mathbf{g}^T H \mathbf{g}$ больше $\varepsilon \mathbf{g}^T \mathbf{g}$. Чтобы понять, страдает ли задача обучения нейронной сети от плохой обусловленности, можно понаблюдать за квадратом нормы градиента $\mathbf{g}^T \mathbf{g}$ и членом $\mathbf{g}^T H \mathbf{g}$. Во многих случаях норма градиента не сильно уменьшается за время обучения, тогда как член $\mathbf{g}^T H \mathbf{g}$ возрастает больше, чем на порядок.

В результате обучение происходит очень медленно, несмотря на большой градиент, т.к. приходится уменьшать скорость обучения, чтобы компенсировать еще большую кривизну.

4.4.2. Локальные минимумы

Одна из самых важных черт *выпуклой оптимизации* состоит в том, что такую задачу можно свести к *задаче нахождения локального минимума*. Гарантируется, что любой *локальный* минимум одновременно является *глобальным*.

У некоторых выпуклых функций в нижней части графика имеется не единственный глобальный минимум, а *целый плоский участок*. Однако любая точка на плоском участке является допустимым решением. При оптимизации выпуклой функции мы точно знаем, что, обнаружив критическую точку любого вида, мы нашли хорошее решение.

У невыпуклых функций, в частности нейронных сетей, локальных минимумов может быть несколько. Более того, почти у любой глубокой модели гарантированно имеется множество локальных минимумов. Впрочем, это не всегда серьезная проблема.

Локальные минимумы становятся проблемой, если значение функции стоимости в них велико, по сравнению со значением в глобальном минимуме. Можно построить небольшую нейронную сеть, даже без скрытых блоков, в которой стоимость в локальных минимумах будет выше, чем в глобальном. Если локальные минимумы с высокой стоимостью встречаются часто, то градиентные алгоритмы оптимизации сталкиваются с серьезной проблемой.

Вопрос о том, много ли локальных минимумов с высокой стоимостью в практически интересных сетях и наталкиваются ли на них алгоритмы оптимизации, остается открытым.

В этой области ведутся активные исследования, но специалисты склоняются к мнению, что для достаточно больших нейронных сетей в большинстве локальных минимумов значение функции стоимости мало и что важно не столько найти глобальный минимум, сколько какую-нибудь точку в пространстве параметров, в которой стоимость низкая, *пусть и не минимальная*.

Чтобы исключить локальные минимумы как возможную причину проблем, имеет смысл построить график зависимости нормы градиента от времени. Если норма градиента не убывает почти до нуля, то проблема не в локальных минимумах и вообще не в критических точках. В пространствах высокой размерности установить с полной определенностью, что корень зла – локальные минимумы, бывает очень трудно. Малые градиенты характерны для многих особенностей строения, помимо локальных минимумов.

4.4.3. Плато, седловые точки и другие плоские участки

Для многих невыпуклых функций в многомерных пространствах локальные минимумы (и максимумы) встречаются гораздо реже других точек с нулевым градиентом: седловых точек.

В седловой точке матрица Гессе имеет как положительные, так и отрицательные собственные значения. Можно считать, что седловая точка является локальным минимумом в одном сечении графика функции стоимости и локальным максимумом – в другом.

Многие классы случайных функций демонстрируют следующее поведение: в пространстве низкой размерности локальные минимумы встречаются часто, а в пространствах большей размерности они редкость, зато часто встречаются седловые точки.

У многих случайных функций есть удивительное свойство: вероятность положительности собственных значений матрицы Гессе возрастает при приближении к областям низкой стоимости. В нашей аналогии с подбрасыванием монеты это означает, что вероятность n раз подряд выкинуть орла выше, если мы находимся в критической точке с низкой стоимостью. Это также означает, что локальные минимумы с низкой стоимостью гораздо вероятнее, чем с высокой. Критические точки с высокой стоимостью с куда большей вероятностью являются седловыми точками. А критические точки с очень высокой стоимостью, скорее всего, являются локальными максимумами.

Каковы последствия изобилия седловых точек для алгоритмов обучения? В случае оптимизации первого порядка, когда используется только информация о градиенте, ситуация неясна. Градиент часто оказывается очень мал в окрестности седловой точки. С другой стороны, есть эмпирические свидетельства в пользу того, что метод градиентного спуска во многих случаях способен выйти из седловой точки.

Для *метода Ньютона седловые точки* представляют очевидную *проблему*. Идея алгоритма градиентного спуска – «спуск с горы», а не явный поиск критических точек. С другой стороны, *методы Ньютона специально предназначен для поиска точек с нулевым градиентом*. Без надлежащей модификации он вполне может найти седловую точку.

Изобилие седловых точек в многомерных пространствах объясняет, почему методы второго порядка не смогли заменить градиентный спуск в обучении нейронных сетей.

4.4.4. Долгосрочные зависимости

Еще одна трудность для алгоритмов оптимизации нейронной сети возникает, когда граф вычислений становится очень глубоким. Такие графы характерны для многослойных сетей прямого распространения, а также для реккурентных сетей, в которых очень глубокий граф вычислений создается в результате применения одной и той же операции на каждом шаге длинной временной последовательности. Повторное применение одних и тех же параметров вызывает особенно трудно преодолимые сложности.

Целевая функция сильно нелинейной глубокой нейронной сети или реккурентной сети часто характеризуется резкими нелинейностями в пространстве параметров, возникающими из-за переменожения нескольких параметров. Когда параметры приближаются к подобному утесу, шаг обновления в методе градиентного спуска может сдвинуть параметры очень далеко, при этом может потеряться все, чего удалось достичь в ходе предшествующей оптимизации.

Например, предположим, что граф вычислений содержит путь, состоящий из повторных умножений на матрцу W. Выполнение t шагов эквивалентно умножению на W^t. Пусть спек-

тральное разложение W имеет вид $W = V\text{diag}(\lambda)V^{-1}$. В этом случае легко видеть, что

$$W^t = (V\text{diag}(\lambda)V^{-1})^t = V\text{diag}(\lambda)^t V^{-1}.$$

Все собственные значения λ_i, кроме близких к 1 по абсолютной величине, либо резко возрастают (если их абсолютная величина больше 1), либо почти обращаются в 0 (если абсолютная величина меньше 1). Когда говорят о *проблеме исчезающего и взрывного градиента*, имеют в виду, что в результате вычислений с таким графом *градиенты* также умножаются на коэффициент $\text{diag}(\lambda)^t$.

Если градиент обращается в 0, то трудно понять, в каком направлении изменять параметры, чтобы улучшить функцию стоимости, а если резко возрастает, то обучение становится численно неустойчивым.

В реккурентных сетях на каждом шаге используется одна и та же матрица W, но в сетях прямого распространения это не так, поэтому даже в очень глубоких сетях прямого распространения, как правило, удается избежать проблемы исчезающего и взрывного градиента.

4.4.5. Плохое соответствие между локальной и глобальной структурами

В центре многих исследований, посвященных трудностям оптимизации, находится вопрос о том, достигает ли обучение глобального минимума, локального минимума или седловой точки, но на практике нейронные сети не находят никакую критическую точку.

В контексте обучения нейронных сетей нас обычно не интересует нахождение точного минимума функции, нужно лишь найти значение, достаточно малое для получения хорошей ошибки обобщения.

4.5. Основные алгоритмы

4.5.1. Стохастический градиентный спуск

Метод стохастического градиентного спуска и его варианты – пожалуй, самые употребительные алгоритмы машинного обучения вообще и глубокого обучения в частности.

Основной параметр алгоритма СГС – скорость обучения. Ранее при описании СГС мы считали скорость обучения ε фиксированной. На практике же необходимо постепенно уменьшать ее со временем, поэтому будем обозначать ε_k скорость обучения на k-ой итерации.

Связано это с тем, что СГС-оценка градиента вносит источник шума (случайная выборка m обучающих примеров), который не исчезает, даже когда мы нашли минимум. Напротив, при использовании *пакетного градиентного спуска* истинный градиент полной функции стоимости уменьшается по мере приближения к минимуму и обращается в 0 в самой точке минимума, так что *скоросто обучения можно зафиксировать*.

Достаточные условия сходимости СГС имеют вид

$$\sum_{k=1}^{\infty} \varepsilon_k = \infty \text{ и } \sum_{k=1}^{\infty} \varepsilon_k^2 < \infty.$$

На практике скорость обучения обычно уменьшают линейно до итерации с номером τ

$$\varepsilon_k = (1-\alpha)\varepsilon_0 + \alpha\varepsilon_\tau,$$

где $\alpha = k/\tau$. После τ-й итерации ε остается постоянным.

Скорость обучения можно выбирать методом проб и ошибок, но обычно лучше понаблюдать за кривыми обучения – зависимостью целевой функции от времени.

4.5.2. Импульсный метод

Стохастический градиентный спуск остается популярной стратегией оптимизации, но обучение с его помощью иногда происходит слишком медленно. *Импульсный метод* (Polyak, 1964) призван ускорить обучение, *особенно в условиях высокой кривизны*, небольших, но устойчивых градиентов или зашумленных градиентов.

В импульсном алгоритме вычисляется *экспоненциально затухающее скользящее среднее прошлых градиентов* и продолжается движение в этом направлении.

Импульсный метод призван решить две проблемы:

- плохую обусловленность матрицы Гессе,
- дисперсию стохастического градиента.

Формально говоря, в импульсном алгоритме вводится переменная v, играющая роль скорости, – это направление и скорость перемещения в пространстве параметров. Скорость устанавливается равной экспоненциально затухающему скользящему среднему градиента со знаком минус. Гиперпараметр $\alpha \in [0, 1)$ определяет *скорость экспоненциального затухания вкладов предшествующих градиентов*. Правило обновления имеет вид

$$\theta \leftarrow \theta + v,\ v \leftarrow \alpha v - \varepsilon \nabla_\theta \Big(\frac{1}{m} \sum_{i=1}^{m} L(f(x^{(i)}; \theta), y^{(i)}) \Big).$$

Раньше размер шага был равен просто норме градиента, умноженной на скорость обучения, т.е. $\varepsilon \|g\|$. Теперь же шаг зависит от величины и сонаправленности предшествующих градиентов. Размер шага максимален, когда много последовательных градиентов указывают точно в одном и том же направлении. Если импульсный алгоритм всегда видит градиент g, то он будет ускоряться в направлении $-g$, пока не достигнет конечной скорости, при которой размер шага равен

$$\frac{\varepsilon \|g\|}{1 - \alpha}.$$

Таким образом, полезно рассматривать гиперпараметр импульса в терминах $1/(1-\alpha)$. Например, $\alpha = 0.9$ соответствует умножению *максимальной скорости* на 10 относительно стандартного алгоритма градиентного спуска.

На практике обычно задают α равным 0.5, 0.9 или 0.99. Как и скорость обучения, α может меняться со временем. Как правило, начинают с небольшого значения и постепенно увеличивают его. Изменение α со временем не так важно, как уменьшение ε со временем.

4.5.3. Метод Нестерова

В работе Sutskever et al. описан вариант импульсного алгоритма, созданный под влиянием метода ускоренного градиента Нестерова. Правила обновления в этом случае имеют вид

$$\theta \leftarrow \theta + v,\ v \leftarrow \alpha v - \varepsilon \nabla_\theta \Big[\frac{1}{m} \sum_{i=1}^{m} L(f(x^{(i)}; \theta + \alpha v), y^{(i)}) \Big],$$

где параметры ε и α играют ту же роль, что и в стандартном импульсном методе. Разница между методом Нестерова и стандартным импульсным методом заключается в точке, где вычисляется градиент. В методе Нестерова градиент вычисляется после применения текущей скорости. Таким образом, метод Нестерова можно интерпретировать как попытку добавить поправочный множитель к стандартному импульсному методу.

В случае выпуклой оптимизации пакетным градиентным спуском метод Нестерова повышает скорость сходимости ошибки превышения с $O(1/k)$ (после k шагов) до $O(1/k^2)$. К сожалению, в случае стохастического градиентного спуска метод Нестерова не улучшает скорость сходимости.

4.6. Стратегии инициализации параметров

Алгоритмы глубокого обучения как правило итеративные и потому пользователь должен указать, с какой точки начинать итерации. Кроме того, обучение глубоких моделей – задача настолько сложная, что большинство алгоритмов сильно зависит от выбора начальных значений. От начальной точки может зависеть, сойдется ли вообще алгоритм, причем некоторые начальные точки так неустойчивы, что алгоритм сталкивается с численными трудностями и завершается ошибкой. Если все-таки алгоритм сходится, то начальная точка определяет скорость сходимости и стоимость в конечной точке – высокую или низкую. Кроме того, для точек с сопоставимой стоимостью ошибка обобщения может различаться очень сильно, и на нее начальная точка тоже может оказывать влияние.

Современные стратегии инициализации просты и основаны на эвристических соображениях. Проектирование улучшенной стратегии инициализации – трудная задача, потому что еще нет отчетливого понимания оптимизации нейронных сетей. Большинство стратегий основано на стремлении получить некоторые полезные свойства сети в начальный момент. Однако мы плохо понимаем, какие из этих свойств и при каких условиях сохраняются после начала процесса обучения. Дополнительная трудность состоит в том, что некоторые начальные точки хороши с точки зрения оптимизации, но никуда не годятся с точки зрения обобщаемости. Наше понимание того, как начальная точка влияет на обобщаемость, совсем уж примитивно, оно не дает почти или вообще никаких указаний на то, как выбирать начальную точку.

Пожалуй, единственное, что мы знаем наверняка, – это то, что начальные параметры должны «*нарушить симметрию*» между разными блоками. Если два скрытых блока с одинаковыми функциями активации соединены с одними и теми же входами, то у этих блоков должны быть разные начальные параметры. Если начальные параметры одинаковы, то детерминированный алгоритм обучения, применяемый к детерминированной функции стоимости и модели, будет всякий раз обновлять эти блоки одинаково.

Даже если модель или алгоритм обучения способны стохастически вычислять разные обновления блоков (например, если при обучении используется прореживание), обычно предпочтительнее инициализировать каждый блок, так чтобы он вычислял свою функцию иначе, чем все остальные блоки. Это позволит гарантировать, что никакие входные паттерны не потеряются в нуль-пространстве прямого распространения, и никакие паттерны градиентов не потеряются в нуль-пространстве обратного распространения. Стремление к тому, чтобы все блоки вычисляли разные функции, диктует выбор в пользу случайной инициализации параметров.

В типичном случае мы выбираем в качестве смещений блоков эвристически выбранные константы, а случайно инициализируем только веса. Дополнительные параметры, например, условная дисперсия предсказания, обычно тоже задаются эвристическими константами.

Почти всегда веса модели инициализируются случайными значениями с номальным или равномерным распределением. Вопрос о том, какое распределение лучше, похоже, не играет особой роли, но тщательно он не изучался. А вот масштаб начального распределения сильно влияет как на результат процедуры оптимизации, так и на способность сети к обобщению.

Чем больше начальные веса, тем сильнее эффект нарушения симметрии, что помогает избежать избыточных блоков. Большие начальные веса помогают также предотвратить потерю сигнала во время прямого или обратного распространения через линейные компоненты каждого слоя – чем больше значения в матрице, тем больше результат умножения матриц.

Однако, если начальные веса слишком велики, то может случиться взрывной рост значений во время прямого или обратного распространения. В *рекуррентных сетях* большие веса также могут привести к *хаосу* (настолько высокой чувствительности к малым возмущениям входного сигнала, что поведение детерминированной процедуры прямого распространения представляется случайным). Проблему взрывного градиента можно в какой-то мере сгладить путем отсечения градиента (сравнения градиента с порогом перед выполнением шага градиентного спуска). Кроме того, большие веса могут стать причиной экстримальных значений, что ведет к насыщению функции активации и полной потере градиента при распространении через насыщенные блоки. Балансирование этих разнонаправленных факторов и определяет идеальный масштаб весов.

Взгляды на проблему с точки зрения регуляризации и оптимизации могут дать совершенно разные подходы к инициализации сети. С точки зрения оптимизации, веса должны быть достаточно большими, чтобы способствовать успешному распространению информации. Но соображения регуляризации побуждают делать веса поменьше.

Использование таких алгоритмов оптимизации, как стохастический градиентный спуск, который производит инкрементные изменения весов и выказывает тенденцию к остановке в областях, близких к начальным параметрам (то ли из-за застревания в области низких градиентов, то ли потому, что сработал критерий ранней остановки вследствие угрозы переобучения), выражает априорное знание о том, что конечные параметры должны быть близки к начальным.

В общем случае градиентный спуск с ранней остановкой – не то же самое, что снижение весов, но между ними можно провести некоторую аналогию, позволяющую рассуждать об эффекте инициализации.

Существуют некоторые эвристики для выбора начального масштаба весов. Одна из них – инициализировать веса в полносвязном слое с m входами и n выходами, выбирая каждый вес из распределения (Glorot and Bengio)

$$W_{i,j} \sim U\Big(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\Big).$$

Эта последняя эвристика выражает компромисс между желанием инициализировать все слои, так чтобы была одинакова дисперсия активации, и желанием инициализировать их, так чтобы была одинакова дисперсия градиента. Формула выведена в предположении, что сеть включает только цепочку умножений матриц *безо всяких нелинейностей*.

Очевидно, что реальные нейронные сети не удовлетворяют этому предположению, но многие стратегии, разработанные для линейных моделей, дают неплохие результаты и в нелинейных сетях.

К сожалению, оптимальные критерии для начальных весов зачастую не приводят к оптимальному качеству. Тому может быть три причины:

- Во-первых, неподходящий критерий – возможно, он не способствует сохранению нормы сигнала во всей сети.
- Во-вторых, свойства, справедливые в момент инициализации, могут нарушаться после начала обучения.
- В-третьих, критерий может ускорять оптимизацию, но непреднамерено увеличивать ошибку обобщения.

На практике масштаб весов обычно следует рассматривать как гиперпараметр, оптимальные значения которого близко к теоретически предсказанному, но не совпадает с ним.

Если вычислительные ресурсы позволяют, обычно имеет смысл рассматривать *начальный масштаб весов* в каждом слое как гиперпараметр и выбирать масштабы, применяя какой-нибудь из алгоритмов поиска гиперпараметров, например, случайный поиск. Или можно вручную поискать наилучшие возможные веса.

Хорошее эврестическое правило выбора начальных масштабов – проанализировать диапазон стандартного отклонения активаций или градиентов на одном мини-пакете данных. Если веса слишком малы, то диапазон активаций будет сужаться по мере прямого распространения по сети. Раз за разом определяя первый слой с неприемлемой малой активацией и увеличивая веса в нем, можно в конце концов получить сеть с разумными начальными активациями снизу доверху. Если в этот момент обучение все еще происходит слишком медленно, то полезно также проанализировать диапазон стандартных отклонений градиентов и активаций. В приниципе, эту процедуру можно автоматизировать, и в общем случае она вычислительно дешевле, чем оптимизация гиперпараметра, основанная на ошибке на контрольном наборе, поскольку в основе лежит обратная связь с поведением начальной модели на одном пакете данных, а не с обученной моделью на контрольном наборе. Этот эвристический протокол используется уже долгое время, но лишь недавно он был формализован и изучен в работе Mishkin and Matas (2015).

До сих пор мы гововрили об инициализации весов. К счастью, инициализация других параметров обычно проще.

Подходы к заданию *смещений* и *весов* должны быть *согласованы*. Инициализация *всех смещений нулями* совместима с большинством схем *инициализации весов*. Есть несколько ситуаций, в которых разумно присваивать некоторым смещениям ненулевые значения:

- Если речь идет о смещении для выходного блока, то часто имеет смысл инициализировать его так, чтобы получилась правильная маргинальная статистика выхода. Для этого предположим, что начальные веса настолько малы, что выход блока определяется только смещением. Это оправдывает выбор в качестве смещения величины, обратной значению функции активации, примененной к маргинальной статистике выхода в обучающем наборе. Например, если выходом является распределение классов, и это распределение сильно скошено, а маргинальная вероятность i-ого класса задается элементом c_i некоторого вектора $\mathbf{c}$, то вектор смещений $\mathbf{b}$ можно найти из уравнения $\mathrm{softmax}(\mathbf{b}) = \mathbf{c}$. Это относится не только к классификаторам, но и к другим моделям – например автокодировщикам и машинам Больцмана. В этих моделях имеются слои, выход которых должен быть похож на вход $\mathbf{x}$, и было бы очень полезно инициализировать смещения таких слоев в соответствии с маргинальным распределением $\mathbf{x}$.
- Иногда мы хотим выбрать смещение так, чтобы предотвратить слишком сильное насыщение на стадии инициализации. Например, мы можем задать смещение скрытого ReLU-блока равным 0.1, а не 0, чтобы избежать его насыщения. Но этот подход несовместим со схемами

инициалзиции весов, которые не ожидают сильного входного сигнала от смещения. Например, его не рекомендуется использовать вместе с инициализацией случайным блужданием.

- Иногда один блок управляет тем, могут ли другие блоки принимать участие в вычислении функции. В таких ситуациях имеются блок с выходом u и другой блок $h \in [0, 1]$, они перемножаются, и на выходе получается uh. Мы можем рассматривать h как вентиль, определяющий, будет ли $uh \approx u$ или $uh \approx 0$. Тогда мы хотим на этапе инициализации задать смещение для h, так чтобы $h \approx 1$ большую часть времени. В противном случае у u не будет возможности обучиться. Например, в работе Jozefowicz et al. (2015) рекомендуется устанавливать смещение 1 для вентиля забывания в модели LSTM.

Еще один распространенный параметр – дисперсия, или точность. Например, мы можем выполнить линейную регрессию с оценкой условной дисперсии с помощью модели

$$p(y|x) = \mathcal{N}(y|\mathbf{w}^T\mathbf{x} + b, 1/\beta),$$

где β – параметр точности. Обычно безопасно инициализировать дисперсию, или точность, значением 1. Другой подход – предположить, что начальные веса настолько близки к нулю, что смещения можно задавать, игнорируя влияние весов, и тогда задать смещения так, чтобы порождалось правильное маргинальное среднее выхода, а дисперсии сделать равными маргниальной дисперсии выхода в обучающем наборе.

4.7. Алгоритмы с адаптивной скоростью обучения

Специлисты по нейронным сетям давно поняли, что скорость обучения – один из самых трудных для установки гиперпараметров, поскольку она существенно влияет на качество модели. Импульсный алгоритм может в какой-то мере сгладить эти проблемы, но ценой введения другого гиперпараметра (коэффициента экспоненциального затухания). Естественно возникает вопрос, нет ли какого-то другого способа. Если мы полагаем, что направления чувствительности почти параллельны осям, то, возможно, имеет смысл задавать *скорость обучения отдельно для каждого параметра* и автоматически адаптировать эти скорости на протяжении всего обучения.

Алгоритм delta-bar-delta – один из первых эвристических подходов к адаптации индивидуальных скоростей обучения параметров модели. Он основан на простой идее: если частная производная функции потерь по данному параметру модели не меняет знак, то скорость обучения следует увеличить. Если же знак меняется, то скорость следует уменьшить. Конечно, такого рода правило применимо только к оптимизации на полном пакете.

NB: В адаптивных методах оптимизации градиенты вычисляются на основе мини-пакетов экземпляров обучающего поднабора данных.

4.7.1. AdaGrad

Алгоритм AdaGrad по отдельности *адаптирует скорости обучения всех параметров модели*, умножая их на коэффициент, обратно пропорциональный квадратному корню из суммы всех прошлых значений квадрата градиента. Для параметров, по которым частная производная функции потерь наибольшая, скорость обучения уменьшается быстро, а если частная производная мала, то и скорость обучения уменьшается медленнее. В итоге больший прогресс получается в направлениях пространства параметров со сравнительно пологими склонами.

В случае выпуклой оптимизации у алгоритма AdaGrad есть некоторые желательные теоретические свойства. Но эмпирически при обучении глубоких нейронных сетей накапливание квадратов градиента с самого начала обучения может привести к преждевременному и чрезмерному уменьшению *эффективной скорости обучения*.

4.7.2. RMSProp

Алгоритм RMSProp (Hinton, 2012) – это модификация AdaGrad, призванная улучшить его поведение в *невыпуклом* случае путем изменения способа агрегирования градиента на экспоненциально взвешенное скользящее среднее. AdaGrad разрабатывался для быстрой сходимости в применении к *выпуклой функции*. Если же он применяется к невыпуклой функции для обучения нейронной сети, то траектория обучения может проходить через много разных структур и в конечном счете прийти в *локально выпуклую впадину*. AdaGrad *уменьшает скорость обучения*, принимая во внимание *всю историю квадрата градиента*, и может случиться так, что скорость станет слишком малой еще до достижения такой выпуклой структуры. В алгоритме RMSProp используется экспоненциально затухающее среднее, т.е. далекое прошлое отбрасывается, чтобы повысить скорость сходимости после обнаружения выпуклой впадины, как если бы внутри этой впадины алгоритм AdaGrad был инициализирован заново.

NOTE: Получается, что RMSProp можно рассматривать как алгоритм поиска локально выпуклых структур, на которых запускается алгоритм AdaGrad, ориентированный на задачи выпуклой оптимизации.

4.7.3. Adam

Adam – еще один алгоритм оптимизации с адаптивной скоростью обучения. Название «Adam» – сокращение от «adaptive moment» (адаптивные моменты). Его, наверное, правильнее всего рассматривать как комбинацию RMSProp и импульсного метода с несколькими важными отличиями. Во-первых, в Adam импульс включен непосредственно в виде оценки первого момента (с экспоненциальными весами) градиента. Самый прямой способ добавить импульс в RMSProp – применить его к масштабированным градиентам. У использования импульса в сочетании с масштабированием нет ясного теоретического обоснования. Во-вторых, Adam включает поправку на смещение в оценки как первых моментов (член импульса), так и вторых (нецентрированных) моментов для учета их инициализации в начале координат. RMSProp также включает оценку (нецентрированного) второго момента, однако в нем нет поправочного коэффициента. Таким образом, в отличие от Adam, в RMSProp оценка второго момента может иметь высокое смещение на ранних стадиях обучения.

Вообще говоря, Adam считается *довольно устойчивым* к выбору гиперпараметров, хотя скорость обучения иногда нужно брать отличной от предлагаемой по умолчанию.

4.7.4. Выбор правильного алгоритма оптимизации

К сожалению, в настоящее время единого мнения нет. Сейчас наиболее популярны и активно применяются алгоритмы СГС, СГС с учетом импульса, RMSProp, RMSProp с учетом импульса, AdaDelta и Adam.

4.8. Приближенные методы второго порядка

Для простоты мы будем рассматривать только одну целевую функцию: эмпирический риск

$$J(\theta) = \mathbb{E}_{\mathbf{x},y\sim\hat{p}_{data}} L(f(\mathbf{x}^{(i)};\theta), y^{(i)}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}^{(i)};\theta), y^{(i)}).$$

4.8.1. Метод Ньютона

В отличие от методов первого порядка, градиентные методы второго порядка для улучшения оптимизации задействуют вторые производные. Самый известный метод второго порядка – метод Ньютона. Опишем его более подробно с акцентом на применение к обучению нейронных сетей.

Метод Ньютона основан на использовании разложения в ряд Тейлора с точностью до членов второго порядка для аппроксимации $J(\theta)$ в окрестности некоторой точки θ_0 производные более выского порядка при этом игнорируются

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^T \nabla_\theta J(\theta_0) + 1/2(\theta - \theta_0)^T H(\theta - \theta_0),$$

где H – гессиан J относительно θ, вычисленный в точке θ_0. Пытаясь найти критическую точку этой функции, мы приходим к *правилу Ньютона для обновления параметров*

$$\theta^* = \theta_0 - H^{-1}\nabla_\theta J(\theta_0). \tag{1}$$

Таким образом, для *локально квадратичной функции* (с положительно определенной матрицей H) умножение градиента на H^{-1} *сразу дает точку минимума*. Если целевая функция выпуклая, но не квадратичная (имеются члены более высокого порядка), то это обновление можно повторить.

Для *неквадратичных* поверхностей *метод Ньютона* можно применять *итеративно*, при условии, что *матрица Гессе* остается *положительно определенной*, $H \succ 0$. Отсюда вытекает двухшаговая итеративная процедура. Сначала мы обновляем или вычисляем обратный гессиан (путем обновления *квадратичной аппроксимации*). Затем обновляем параметры в соответствии с формулой (1).

Метод Ньютона применим, только если *матрица Гессе положительно определена* (т.е. все собственные значения матрицы Гессе положительны). В *глубоком обучении* поверхность целевой функции обычно *невыпуклая* и имеет много особенностей *типа седловых точек*, с которыми метод Ньютона не справляется. Если *не все собственные значения матрицы Гессе положительны*, например вблизи седловой точки, то метод Ньютона может производить обновление *не в том направлении*. Такую ситуацию можно предотвратить с помощью *регуляризации гессиана*. Одна из распространненых стратегий регуляризации – прибавление константны α ко всем диагональным элементам гессиана. Тогда *регуляризованное обновление* принимает вид

$$\theta^* = \theta_0 - \left[H(f(\theta_0)) + \alpha I\right]^{-1}\nabla_\theta f(\theta_0).$$

Эта стратегия регуляризации применяется в аппроксимации метода Ньютона, например в *алгоритме Левенберга-Марквардта*, и работает неплохо, если *отрицательные собственные значения гессиана сравнительно близки к нулю*. Если же в некоторых направлениях кривизна сильнее, то значение α следует выбирать достаточно большим для компенсации отрицательных собствен-

ных значений. Однако по мере увеличения α в гессиане начинает доминировать диагональ αI, и направление, выбранное методом Ньютона, стремится к стандартному градиенту, поделенному на α.При наличии сильной кривизны α должно быть настолько большим, чтобы метод Ньютона делал меньшие шаги, чем градиентный спуск с подходящей скоростью обучения.

Помимо проблем, связанных с такими особенностями целевой функции, как *седловые точки*, применение метода Ньютона к обучению больших нейронных сетей лимитируется требованиями к вычислительным ресурсам. Число элементов гессиана равно квадрату числа параметров, поэтому при k параметрах (а даже для совсем небольшой нейронной сети параметры могут исчисляться миллионами) *метод Ньютона требует обращения матрицы размера* $k \times k$, а вычислительная сложность этот операции составляет $O(k^3)$. Кроме того, поскольку параметры изменяются при каждом обновлении, *обратный гессиан* нужно вычислять *на каждой итерации обучения*.

4.8.2. Метод сопряженных градиентов

Метод сопряженных градиентов позволяет *избежать вычисления обратного гессиана* посредством итеративного спуска в *сопряженных направлениях*.

Метод наискорейшего спуска в квадратичной впадине неэффективен, т.к. продвигается зигзагами. Так происходит, потому что направление линейного поиска, определяемое на очередном шаге, *гарантировано ортогонально* направлению поиска на предыдущем шаге.

Выбор ортогональных направлений спуска не сохраняет минимума вдоль предыдущих направлений поиска. Отсюда и зигзагообразная траектория, поскольку после спуска к минимуму в направлении текущего градиента мы должны заново минимизировать целевую функцию в направлении предыдущего градиента. А значит, следуя направлению градиента в конце каждого отрезка, мы в некотором смысле перечеркиваем достигнутое в направлении предыдущего отрезка. Метод сопряженных градиентов и призван решить эту проблему.

В методе сопряженных градиентов направление следующего поиска является сопряженным к направлению предыдущего, т.е. мы не отказываемся от того, что было достигнуто в предыдущем направлении. На t-ой итерации обучения направление следующего поиска определяется формулой

$$d_t = \nabla_\theta J(\theta) + \beta_t d_{t-1},$$

где β_t – коэффициент, определяющий, какую часть направления d_{t-1} следует прибавить к текущему направлению поиска.

Два направления d_t и d_{t-1} называются *сопряженными*, если $d_t^{\mathrm{T}} H d_{t-1} = 0$, где H – матрица Гессе.

Самый простой способ обеспечить сопряженность – вычислить собственные векторы H для выбора β_t – не отвечает исходной цели разработать метод, который был бы вычислительно проще метода Ньютона при решении больших задач. Существует два популярных метода вычисления β_t

1. Метод Флетчера-Ривса

$$\beta_t = \frac{\nabla_\theta J(\theta_t)^T \nabla_\theta J(\theta_t)}{\nabla_\theta J(\theta_t)^T \nabla_\theta J(\theta_t)}$$

2. Метод Полака-Рибьера

$$\beta_t = \frac{(\nabla_\theta J(\theta_t) - \nabla_\theta J(\theta_{t-1}))^T \nabla_\theta J(\theta_t)}{\nabla_\theta J(\theta_{t-1})^T \nabla_\theta J(\theta_{t-1})}$$

Для *квадратичной* поверхности сопряженность направлений гарантирует, что модуль градиента вдоль предыдущего направления не увеличивается. Поэтому минимум, найденный вдоль предыдущих направлений, сохраняется. Следовательно, в k-мерном пространстве параметров метод сопряженных градиентов требует не более k поисков для нахождения минимума.

Нелинейный метод сопряженных градиентов. До сих пор мы рассматривали метод сопряженных градентов в применении к квадратичной целевой функции. Но нас в основном интересуют методы оптимизации для обучения нейронных сетей и других глубоких моделей, в которых целевая функция далека от квадратичной. Как ни странно, метод сопряженных градиентов применим и в такой ситуации, хотя с некоторыми изменениями. Если целевая функция *не квадратичная*, то *уже нельзя гарантировать*, что поиск в сопряженном направлении сохраняет минимум в предыдущих направлениях. Поэтому в *нелинейном алгоритме сопряженных градиентов* время от время производится *сброс*, когда метод сопряженных градиентов заново начинает поиск вдоль направляений неизменного градиента.

Есть сообщения, что нелинейный метод сопряженных градиентов дает неплохие результаты при обучении нейронных сетей, хотя часто имеет смысл инициализировать оптимизацию, выполнив несколько итераций стохастического градиентного спуска, и только потом переходить к нелинейным сопряженным градиентам. Кроме того, хотя нелинейный алгоритм сопряженных градиентов традиционно считался пакетным методом, его мини-пакетные варианты успешно применялись для обучения нейронных сетей. Позже были предолжены адаптации метода сопряженных градиентов, например алгоритм масштабированных сопряженных градиентов.

4.8.3. Алгоритм BFGS

Алгоритм Бройдена-Флетчера-Гольдфарба-Шанно – попытка взять некоторые преимущества метода Ньютона без обременительных вычислений. В этом смысле BFGS аналогичен методу сопряженных градиентов. Однако в BFGS подход к аппроксимации обновления Ньютона более прямолинейный. Обновление Ньютона определяется формулой

$$\theta^* = \theta_0 - H^{-1} \nabla_\theta J(\theta_0),$$

где H – гессиан J относительно θ, вычисленный в точке θ_0.

Основная вычислительная трудность при применении обновления Ньютона – вычисление обратного гессиана H^{-1}. В *квазиньютоновских методах* (из которых алгоритм BFGS самый известный) *обратный гессиан аппроксимируется матрицей* M_t, которая итеративно уточняется в ходе обновлений низкого ранга.

После нахождения аппроксимации гессиана M_t направление спуска ρ_t определяется по формуле $\rho_t = M_t g_t$. В этом направлении производится линейный поиск для определения величины шага ε^*. Окончательное обновление параметров производится по формуле

$$\theta_{t+1} = \theta_t + \varepsilon^* \rho_t.$$

Как и в методе сопряженных градиентов, в алгоритме BFGS производится последовательность линейных поисков в направлениях, вычисляемых с учетом информации второго порядка. Но, в отличие от метода сопряженных градиентов, успех не так сильно зависит от того, находит ли линейный поиск точку, очень близкую к истинному минимуму вдоль данного направления.

Поэтому BFGS тратит меньше времени на уточнение результатов каждого линейного поиска. С другой стороны, BFGS *должен хранить обратный гессиан* M, для чего требуется память объема $O(n^2)$, поэтому *BFGS непригоден* для современных моделей глубокого обучения, насчитывающих миллионы параметров.

BFGS в ограниченной памяти (L-BFGS). Потребление памяти в алгоритме BFGS можно значительно уменьшить, *если не хранить полную аппроксимацию обратного гессиана* M. В алгоритме L-BFGS аппроксимация M вычисляется так же, как в BFGS, но, вместо того чтобы сохранять аппроксимацию между итерациями, делается предположение, что $M^{(t-1)}$ – единичная матрица. При использовании совместно с точным линейным поиском направления, вычисляемые алгоритмом L-BFGS, являются взаимно сопряженными. Однако, в отличие от метода сопряженных градиентов, эта процедура ведет себя хорошо, даже когда линейный поиск находит только приближенный минимум. Описанную стратегию L-BFGS без запоминания можно обобщить, включив больше информации о гессиане; для этого нужно хранить некоторые векторы, используемые для обновления M на каждом шаге, тогда потребуется только память объмом $O(n)$.

4.9. Стратегии оптимизации и метаалгоритмы

Многие методы оптимизации – не совсем алгоритмы, а скорее общие шаблоны, которые можно специализировать и получать алгоритмы или подпрограммы, влкючаемые в различные алгоритмы.

4.9.1. Пакетная нормировка

Пакетная нормировка – одна из наиболее интересных новаций в области оптимизации глубоких нейронных сетей – вообще алгоритмом не является. Это *метод адаптивной перепараметризации*, появившийся из-за трудностей обучения очень глубоких моделей.

Для очень глубоких моделей характерна композиция нескольких функций, или слоев. Градиент говорит, *как обновлять каждый параметр* в предположении, что *другие слои не изменяются*. На практике мы *обновляем все слои одновременно*. При выполнении обновления могут произойти неожиданности, потому что ко всем образующим композицию функциям одновременно применяются обновления, вычисленные в предположении, что прочие функции сохраняют постоянство.

Пакетная нормировка предлагает элегантный способ перепараметризации почти любой глубокой сети. Перепараметризация значительно снижает остроту проблемы координации обновлений между многими слоями. Пакетную нормировку можно применить к входному и любому скрытому слою сети. Пусть H – *мини-пакет активаций* нормируемого слоя, представленный в виде матрицы плана, так что активации для каждого примера занимают одну строку матрицы. Для нормировки заменим H матрицей

$$H' = \frac{H - \mu}{\sigma},$$

где μ – вектор средних всех блоков, а σ – вектор стнадартных отклонений всех блоков.

Приведенная выше формула выражает применение векторов μ и σ к каждой строке матрице H. Внутри каждой строки операция применяется поэлементно, т.е. для нормировки $H_{i;j}$ нужно вычесть μ_j и разделить на σ_j. Остальная сеть работает с H' точно так же, как исходная работала с H.

На этапе *обучения*

$$\mu = \frac{1}{m}\sum_i H_{i;:}$$

и

$$\sigma = \sqrt{\delta + \frac{1}{m}\sum_i (H-\mu)_i^2},$$

где δ – небольшое положительное число, например 10^{-8}, введенное, чтобы избежать неопределенного градиента $\sqrt{z}$ в точке $z = 0$. И важнейший момент – *мы выполняем обратное распространение сквозь эти операции*, чтобы вычислить среднее и стандартное отклонение и применить их к нормировке H. Это означает, что градиент никогда не предлагает операцию, действие которой сводится просто к увеличению стандартного отклонения или среднего h_i; операции нормировки устраняют результат такого действия и обнуляют его компоненту в градиенте. Это и было главным нововведением пакетной нормировки.

В прежних подходах к функции стоимости прибавлялись штрафы, направленные на то, чтобы блоки имели нормированные статистики активации, или после каждого шага градиентного спуска производилось вмешательство с целью перенормировать статистики блока. Первый подход обычно приводил к неидеальной нормировке, а последний – к значительным затратам времени впустую, потому что алгоритм обучения раз за разом предлагает изменить среднее и дисперсию, а на шаге нормировки это изменение отменяется. Пакетная нормировка перепараметризует модель, так что некоторые блоки всегда стандартизованы по определению, и тем самым ловко обходит обе проблемы.

На этапе *тестирования* μ и σ можно заменить *скользящими средними*, подготовленными на этапе *обучения*. Это позволяет вычислять модель для одного примера, не прибегая к определениям μ и σ, которые зависят от всего мини-пакета.

Действие пакетной нормировки направлено на стандартизацию только среднего и дисперсии каждого блока с целью стабилизировать обучение, но она не препятствует изменению связей между блоками и нелинейных статистик одного блока.

Нормировка среднего и стандартного отклонения блока может снизить выразительную мощность нейронной сети, содержащей этот блок. Для сохранения выразительной мощности обычно заменяют пакет активаций скрытых блоков H на $\gamma H' + \beta$, а не просто на нормированную матрицу H'. Переменные γ и β – обученные параметры, благодаря которым новая величина может иметь произвольные среднее и стандартное отклонение. На первый взгляд, это кажется бессмысленным – зачем было устанавливать среднее в 0, а потом вводить параметр, который позволяет снова переустановить его в произвольное значение β? Да затем, что новая параметризация может представить то же самое семейство функций от входных данных, что и старая, но при этом обладает другой динамикой обучения. В старой параметризации среднее H определялось сложным взаимодействием между параметрами на уровнях ниже H. В новой же параметри-

зации среднее $\gamma H' + \beta$ определяется только величиной β. При новой параметризации модель *гораздо легче обучить методом градиентного спуска.*

Большинство слоев нейронной сети имеет вид $\varphi(XW + b)$, φ – фиксированная нелинейная функция активации, например, преобразование линейной ректификации. Естественно спросить, следует ли применять пакетную нормировку ко входу X или к уже преобразованному значению $XW + b$, В работе Ioffe and Szegdy (2015) рекомендуется последнее. Точнее говоря, $XW + b$ следует заменить результатом нормировки XW. Член смещения нужно опустить, потому что он становится избыточным ввиду параметра β, применяемого в ходе перепараметризации. Входом слоя обычно является выход нелинейной функции активации (например, ReLU) предыдущего слоя. Поэтому статистика входа сильнее отличается от нормального и хуже поддается стандартизации посредством линейных операций. В сверточных сетях важно применять одну и ту же нормировку μ и σ в каждой точке пространства в карте признаков, чтобы статистика карты признаков оставалась одинаковой вне зависимости от положения в пространстве.

4.9.2. Покоординатный спуск

В некотрых случаях задачу оптимизации можно быстро решить, разбив на отдельные части. Если минимизировать $f(\mathbf{x})$ по переменной x_i, затем по переменной x_j и т.д., перебрав в цикле все переменные, то мы *гарантированно* окажемся в (*локальном*) минимуме. Такой подход называется *покоординатным спуском*, поскольку в каждый момент времени производится оптимизация по одной координате.

В более общем случае *блочно-покоординантого спуска* одновременно производится минимизация по *подмножеству переменных*. Термин «покоординатный спуск» часто употребляется не только в своем строгом смысле, но и для обозначения блочно-покоординантного спуска.

Покоординатный спуск наиболее осмыслен, когда различные переменные в задаче оптимизации можно разбить на группы с относительно изолированными ролями или когда оптимизация по одной группе переменных значительно эффективнее, чем по всем. Рассмотрим, к примеру, такую функцию стоимости

$$J(H, W) = \sum_{i,j} |H_{i,j}| + \sum_{i,j} (X - W^T H)^2_{i,j}.$$

Эта функция описывает проблему обучения, которая называется разреженным кодированием. Цель состоит в том, чтобы найти матрицу весов W, которая может линейно декодировать матрицу значений активации H и реконструировать обучающий набор X. В большинстве применений разреженного кодирования участвует также снижение весов или ограничение на нормы столбцов W, чтобы предотвратить патологические решения с очень малой H и большой W.

Функция J невыпуклая. Однако мы можем разбить входы алгоритма обучения на два множества: словарные параметры W и представление кода H. Минимизация целевой функции по любому из этих двух множеств – выпуклая задача.

Поэтому метод блочно-покоординатного спуска позволяет использовать эффективные алгоритмы выпуклой оптимизации, выполнив сначала оптимизацию по W с фиксированным H, а затем оптимизацию по H с фиксированным W.

Покоординатный спуск – не самая удачная стратегия, когда значение одной переменной сильно влияет на оптимальное значение другой, как в функции $f(\mathbf{x}) = (x_1 - x_2)^2 + \alpha(x_1^2 + x_2^2)$, где α – положительная постоянная. Для минимизации первого члена нужно, чтобы переменные мало

отличались друг от друга, а для минимизации второго – чтобы обе были близки к нулю. Минимум достигается, когда обе переменные равны 0. Методом Ньютона эту задачу можно было бы решить *за один шаг* (!), потому что она *квадратичная* и *положительно определенная*. Однако при малых α метод покоординатного спуска сходится очень медленно, потому что первый член не дает изменить одну переменную, так чтобы ее значение сильно отличалось от значения второй переменной.

5. Сверточные сети

Сверточная сеть (LeCun, 1989) – это специальные вид нейронной сети для обработки данных с сеточной топологией. Примерами могут служить временные ряды, которые можно рассматривать как одномерную сетку примеров, выбираемых через регулярные промежутки времени, а также изображения, рассматриваемые как двумерная сетка пикселей.

5.1. Операция свертки

В самом общем виде свертка – это операция над двумя функциями вещественного аргумента. Чтобы получить менее зашумленную оценку положения корабля, необходимо усреднить несколько результатов измерений. Разумеется, недавние измерерния более важны, поэтому мы хотим вычислять взвешенное среднее, придавая недавним измерениям больший вес. Для этого можно воспользоваться весовой функцией $w(a)$, где a – давность измерения. Применив такую операцию усреднения в каждый момент времени, мы получим новую функцию, которая дает *сглаженную оценку* положения космического корабля

$$s(t) = \int x(a)w(t-a)da.$$

Эта операция называется *сверткой* и обычно обозначается звездочкой

$$s(t) = (x * w)(t)$$

В общем случае свертку можно определить для любых функций, для которых определен показанный выше интеграл, и использовать не только лдя получения взвешенного среднего.

В терминологии сверточных сетей первый аргумент (в нашем примере функция x) называется *входом*, а второй (функция w) – *ядром*. Выход иногда называют *картой признаков*.

Дискретная свертка

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a).$$

На практике сумму от минус до плюс бесконечности можно заменить суммированием по конечному числу элементов массива.

Наконец, мы часто используем *свертки по нескольким осям*. Например, если входом является двумерное изображение I, то и ядро должно быть двумерным

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n).$$

Операция свертки *коммутативна*, поэтому формулу можно записать так

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n).$$

Обычно вторую формулу проще реализовать в библиотеке машинного обучения, поскольку диапазон допустимых значений m и n меньше.

Свойство коммутативности свертки имеет место, потому что мы отразили ядро относительно входа, т.е. при увеличении m индекс входа увеличивается, а индекс ядра уменьшается. Единственная причина такого отражения – обеспечить коммутативность. И хотя коммутативность полезна для доказательства теорем, в реализации нейронных сетей она обычно роли не играет. Вместо этого во многих библиотеках реализована родственная функция – *перекрестная корреляция* – та же свертка, только *без отражения ядра*

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n).$$

Во многих библиотеках машинного обучения реализована именно перекрестная корреляция, но называется она сверткой.

5.2. Мотивация

Со сверткой связанны три важные идеи, которые помогают улучшить систему машинного обучения: *разреженные взаимодействия*, *разделение параметров* и *эквивариантные представления*.

В слоях традиционной нейронной сети применяется умножение на матрицу параметров, в которой взаимодействие между каждым входным и каждым выходным блоками описывается отдельным параметром. Это означает, что каждый выходной блок взаимодействует с каждым входным блоком. Напротив, в *сверточных сетях взаимодейтсвия* обычно *разреженные* (это свойство называют еще разреженной связностью, или разреженными весами). Достигается это за счет того, что ядро меньше входа. Например, входное изображение может содержать тысячи или миллионы пикселей, но небольшие значимые признаки, например границы, можно обнаружить с помощью ядра, охватывающего всего десятки или сотни пикселей. Следовательно нужно хранить меньше параметров, а это снижает требования модели к объему памяти и повышает ее статистическую эффективность. Кроме того, для вычисления выхда потребуется меньше операций.

В случае свертки специальный вид разделения параметров наделяет слой свойством, которое называется *эквивариантностью* относительно *параллельного переноса*. Говорят, что функция эквивариантна, если при изменении входа выход меняется точно так же.

Функция $f(x)$ *эквивариантна* относительно функции g, если $f(g(x)) = g(f(x))$. В случае свертки, если g – параллельный перенос, или сдвиг входа, то функция свертки эквивариантна относительно g.

Если перенести событие на более поздний момент времени во входных данных, то на выходе появится точно такое же его представление, только позже. А при работе с изображениями свертка создает двумерную карту появления определенных признаков во входном изображении. Если переместить объект во входном изображении, то его представление на входе переместится на такую же величину.

Это бывает нужно, когда мы знаем, что некоторая функция от небольшого числа пикселей полезна при применении к нескольким участкам входа. Например, в случае обработки изображений полезно обнаруживать границы в первом слое сверточной сети. Одни и те же границы встречаются более-менее везде в изображении, поэтому имеет смысл разделять параметры по всему изображению. Но в некоторых случаях такое глобальное разделение параметров нежелательно. Например, если мы обрабатываем изображения, которые были кадрированы, так чтобы в центре оказалось лицо человека, то, наверное, хотим выделять разные признаки в разных точках – часть сети будет обрабатывать верхнюю часть лица в поисках бровей, а другая часть – искать подбородок в нижней части лица.

Свертка – чрезвычайно эффективный способ описания преобразований, в которых одно и то же линейное преобразование многократно применяется к небольшим участкам изображения.

Свертка не эквивариантна относитльно некоторых других преобразований, например масштабирования или поворота.

5.3. Пулинг

Типичный *слой сверточной сети* состоит из трех стадий:

1. Слой параллельно выполняет несколько *сверток* и порождает множество *линейных активаций*.
2. Каждая *линейная активация* пропускается через *нелинейную функцию активации*, например функцию линейной ректификации. (Детекторная стадия)
3. Используется *функция пулинга* для дальнейшей модификации выхода слоя.

Функция пулинга заменяет выход сети в некоторой точке сводной статистикой близлежащих выходов. Например, операция *max-пулинг* (Zhou and Chellappa, 1988) возвращает максимальный выход в прямоугольной окрестности. Из других употребительных функций пулинга отметим усреднение по прямоугольной окрестности, L_2-норму в прямоугольной окрестности и взвешенное среднее с весами, зависящими от расстояния до центрального пикселя.

В любом случае *пулинг* позволяет сделать представление приблизительно *инвариантным* относительно *малых параллельных переносов входа*. Инвариантность относительно параллельного переноса означает, что если сдвинуть вход на небольшую величину, то значения большинства подвергнутых пулингу выходов не изменятся.

Локальная инвариантность относительно параллельного переноса полезна, если нас больше интересует сам факт существования некоторого признака*, а не его точное местонахождение*. Например, когда мы хотим определить, присутсвтует ли в изображении лицо, нам не важно положение глаз с точностью до пикселя, нужно только знать, есть ли глаз слева и глаз справа. В других ситуациях важнее сохранить местоположение признака. Например, если мы ищем угловую точку, образованную пересечением двух границ, ориентированных определенным образом, то необходимо сохранить положение границ настолько точно, чтобы можно было проверить, пересекаются ли они.

5.4. Свертка и пулинг как бесконечно сильное априорное распределение

Априорное распределение может быть сильным или слабым в зависимости от концентрации плотности вероятности. *Слабым* называется *априорное распределение* с высокой энтропией, например нормальное распределение с большой дисперсией. При таком априорном распределении

параметры могут сдвигаться в зависимости от данных более или менее свободно. У *сильного априорного распределения* очень низкая энтропия, как, например, у нормального распределения с малой дисперсией. Такое распределение играет более активную роль в определении конечных параметров.

В *бесконечно сильном априорном распределении* вероятность некоторых параметров *нулевая*, т.е. утверждается, что такие значения параметров запрещены вне зависимости от того, поддерживаются они данными или нет.

Сверточную сеть можно представлять себе как *полносвязную сеть* с *бесконечно сильным априорным распределением весов*. Оно говорит, что веса некоторого скрытого слоя должны быть идентичны весам соседнего с ним слоя, но сдвинуты в пространстве. Априорное распределение говорит также, что веса должны быть равны 0 всюду, кроме *малого рецептивного поля*, состоящего из смежных блоков, поставленных в соответствие данному скрытому блоку.

Короче говоря, можно считать, что *свертка* вводит бесконечно сильное априорное распределение вероятности параметров слоя, согласно которому обучаемая данным слоем функция допускает только локальные взаимодействия и *эквивариантна относительно параллельных переносов*. Аналогично пулинг вводит бесконечно сильное априорное распределение, согласно которому каждый блок должен быть инваринтен относительно малых параллельных переносов.

Одно из ключевых открытий в том, что *свертка* и *пулинг* могут стать причиной *недообучения*. Как любое априорное распределение, свертка и пулинг полезны, только когда предположения, выравженные этим распределением, достаточно верны. Если в задаче необходимо сохранять точную пространственную информацию, то применение пулинга по всем признакам может увеличить ошибку обучения.

Некоторые архитектуры сверточных сетей рассчитаны на применение пулинга только к части каналов, чтобы получить как признаки с высокой степенью инвариантности, так и признаки, не подверженные недообучению в случае, когда априорное предположение об инвариантности относительно параллельных переносов неверно. Если задача подразумевает включение в состав входных данных информации из очень отдаленных областей, то априорное распределение, ассоциировуемое со сверткой, может оказаться непригодным.

6. Моделирование последовательностей. Рекуррентные и рекурсивные сети

Рассмотрим классическую рекуррентную форму динамической системы

$$s^{(t)} = f(s^{(t-1)}; \theta),$$

s – состояние системы, а θ – вектор параметров модели.

Для конечного числа временных шагов τ граф можно развернуть, применив это определение $(\tau - 1)$ раз.

Например, если развернуть это выражение для $\tau = 3$ шагов, то получим

$$s^{(3)} = f(s^{(2)}; \theta) = f(f(s^{(1)}; \theta); \theta)$$

После такой развертки путем повторного применения определения получается выражение, не содержащее рекурсии. Одни и те же параметры θ используются на всех временных шагах!

Замечание

В рекуррентной сети одни и те же веса разделяются между несколькими временными шагами

Развернутое рекуррентное выражение после t шагов можно представить так

$$h^{(t)} = g^{(t)}(x^{(t)}, x^{(t-1)}, \ldots, x^{(1)}) = f(h^{(t-1)}, x^{(t)}; \theta)$$

Получается, что обучать можно одну модель f, которая действует на всех временных шагах и для последовательностей любой длины, не прибегая к обучению отдельных моделей $g^{(t)}$ для каждого временного шага.

Для вычисления градиента необходимо выполнить *прямое распространение*, двигаясь слева направо по развернутому графу, а затем *обратное распространение*, двигаясь справа налево по тому же графу. Время работы имеет порядок $O(\tau)$, и его нельзя уменьшить за счет распараллеливания, потому что *граф прямого распространения* приниципиально последовательный: каждый временной шаг можно обсчитать только после завершения предыдущего. *Состояния*, вычисленные во время прямого прохода, *необходимо хранить* до повторного использования на обратном проходе, поэтому объем потребляемой памяти также имеет порядок $O(\tau)$. Таким образом, сеть *с рекурсией между скрытыми блоками* является очень мощной, но дорогой для обучения [1, стр. 323].

6.1. Вычисление градиента в рекуррентной нейронной сети

Нужно просто применить обобщенный алгоритм обратного распространения к развернутому графу вычислений. Никаких специальных алгоритмов не нужно. Градиенты, полученные в результате обратного распространения, можно затем использовать в сочетании с любым универсальным градиентным методом для обучения РНС.

6.2. Проблема долгосрочных зависимостей

Основная трудность состоит в том, что градиенты, распространяющиеся через много слоев, либо *исчезают* (в большинстве случаев), либо начинают *взрывообразно расти* (редко, но с большим уроном для оптимизации) [1, стр. 339].

Список литературы

1. *Гудфеллоу Я.* Глубокое обучение, 2018. – 652 с.
2. *Рамальо Л.* Python – к вершинам мастерства: Лаконичное и эффективное программирование. – М.: МК Пресс, 2022. – 898 с.
3. *Хейдт М., Груздев А.* Изучаем pandas. – М.: ДМК Пресс, 2019. – 682 с.